IEEE MICRO

AUGUST 1985

FERMTOR:

A Tunable Multiprocessor Architecture

and other articles on multiprocessing

Also:

High-level Languages

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE MICRO

1986 • EDITORIAL • CALENDAR

February

Semicustom Chip Technology

April

32-Bit Peripheral Chips

June

Telecommunications and Networking

August

Operating Systems

October

Multiprocessing

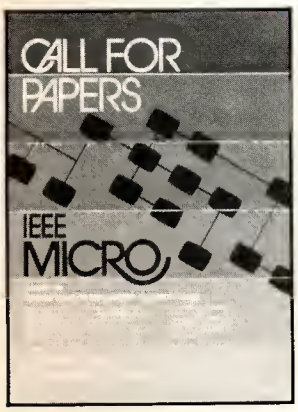
December

Digital Signal Processing

Would you like to submit an article, review submissions, or contribute departmental material for one of IEEE Micro's 1986 issues? Fill out the Reader Interest Card at the back of the magazine.

IEEE MICRO

Volume 5 Number 4 (ISSN 0272-1732) August 1985



On the cover

The FERMTOR tunable multiprocessor architecture. Article on page 5. Inset: The Systech DCP8804 communications board. See page 94.

Cover: Jay Simpson and Larry Keiser. Piano courtesy of Isabelle Gossard.

In the next issue: The Engineering Workstation—Probing the State of the Art.

Departments

Features

Multiprocessing

High-Level Languages

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the IEEE Computer Society: IEEE Headquarters, 345 East 47th St., New York, NY 10017; IEEE Computer Society West Coast Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720. Annual subscription: $12.00 in addition to IEEE Computer Society or any other IEEE society member dues. Nonmember prices: available on request. Single-copy prices: members $7.50; nonmembers $15.00. This journal is also available in microfiche form.

Undelivered copies: Send to 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

Postmaster: Send address changes to *IEEE Micro*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. Second class postage is paid at New York, NY, and at additional mailing offices.

From the Associate Editor-in-Chief

With this message I am pleased to introduce myself to the readers of *IEEE Micro* and to thank the selection committee for appointing me to the position of associate editor-in-chief. *IEEE Micro* now serves a vital role among the range of journals that come across my desk (about 10 to 12 per month), and I hope that in the future my contribution, along with the contributions of the entire editorial board, will make it an indispensible publication for every practicing engineer, manager, and teacher in the microprocessor field.

Because *IEEE Micro* is designed to be at the leading edge of technology, some might assume that its appeal would be limited only to a select few. I do not believe that is necessarily true, however. Our focus is upon a dynamic, interesting, and high-impact area, and there are a great many engineers, computer scientists, and other technology users whose work involves microcomputers and microprocessors in some capacity. Thus, the potential readership of *Micro* is quite broad-based.

The IEEE has always been known for the quality and quantity of its publications. For *Micro* to hold its own in the competitive world of publication, it must continually produce a high-quality product with general appeal. *Micro* has the unique opportunity to become both a journal at the leading edge of the technology and a publication that effectively transfers information about the technology to its readers. The transfer of knowledge from a medium such as a journal to an individual is, at best, a poorly defined science and a true art. It is a major accomplishment to present knowledge in a form that allows individuals to really study, assimilate, and apply it.

Having spent some 20 years in an academic environment, I have observed that learning is a topic-dependent, individual process. If this observation is true, then it seems to follow that efficient mass transfer of information is extremely difficult, if not impossible. Nevertheless, certain texts and papers have been able to bridge the gap of individual learning styles to successfully appeal to the learning interests of the majority of individuals. These classical presentations tend to have two things in common. First, the presentation reduces a difficult concept or theory to understandable terms; and second, the publication occurs early in the development of a theory or concept. Striving to be at the forefront of technology, *IEEE Micro* is in an excellent position to publish several such "classic" papers over the next few years. I feel that our magazine can and should strive for this type of excellence. Perhaps in the future, time and space will permit me to expand on some of these ideas.

Regards,

Joe Hootman

Joe Hootman

Joe Hootman is a professor of electrical engineering at the University of North Dakota in Grand Forks, North Dakota. He has a current interest in microcomputers, signal processing, and CAD systems. He received his BS degree in electrical engineering in 1959 from the University of Missouri at Rolla and his MS and PhD degrees from Iowa State University in 1962 and 1965, respectively. He taught at Colorado State University before assuming his present position, and has had industrial experience with Collins Rockwell and the National Bureau of Standards. He is a member of ASEE, Tau Beta Pi, Eta Kappa Nu, Sigma Xi and the IEEE Computer Society.

Hootman may be contacted at PO Box 7165, Electrical Engineering Department, University of North Dakota, Grand Forks, ND 58202.

Reader Interest

Dear Readers,

June marked the first month when *IEEE Micro* instituted its Reader Interest Card, your direct pipeline to the magazine's editorial board and staff. To those of you who took the time to write in your comments on the card, thank you very much. Your inputs are read and considered. This is your magazine. Your opinion matters.

The reader balloting for the "Best Article of 1984" was not conclusive and did not select a clear-cut winner. We also did not get enough responses for the ballot to be statistically significant. The matter was brought in July before the assembled *IEEE Micro* Editorial Board at NCC in Chicago; and by secret ballot (with me abstaining) the board selected "The Motorola MC68020," by Doug MacGregor, Dave Mothersole, and Bill Moyer (August 1984). The vote agreed with the reader poll, which was very close. Congratulations to all three authors.

Our reader poll from the June issue yielded several "attapersons" and a couple of constructive criticisms. They are all equally welcome. Those who thought the issue and/or format was good to excellent were: W.B. (Panama City, FL); A.W. (Lewisburg, PA); B.H. (Albuquerque, NM); S.S. (NY,NY); C.K. (Sunnyvale, CA); K.M. (Cranston, RI) and several others who just commented "excellent." R.M. (Rolla, MO) felt the previous format was more effective.

By far the most popular article of the issue was Chuck Hastings' piece on "Second-sourcing CPUs...". D.B. (Houghton, MI) felt "Motorola's Silver Quill Program" was an inappropriate topic. W.B. (Panama City, FL) liked VLSI Packaging, the Hastings article, and MicroStandards. Of the columns, MicroStandards and MicroLaw were the hands-down favorites.

If you did not see your initials above because I have lumped your comments with similar ones, please forgive me. All comments are seriously considered.

Regards,

Jim Farrell

To the Editor

Thompson award winners respond

(Editor's note: The authors of the following letter were recently awarded the IEEE Browder J. Thompson Memorial Prize Award for the best article published in 1983 in an IEEE publication by an author or authors under the age of 31. The article, "Virtual Memory and the MC68020," appeared in the June 1983 issue of *IEEE Micro*. Along with coauthor Bill Moyer, MacGregor and Mothersole were voted best *IEEE Micro* authors of 1984 for their August article, "The Motorola MC68020.")

To the Editor:.

We would like to thank the IEEE for awarding us the Browder J. Thompson Memorial Award. While we are thankful for the recognition, we do not feel that we really deserve this award. As is often the case when individuals are selected to receive an award, there are so many deserving who go unnoticed. Instead of considering this award as recognition for our efforts in writing this article, we feel that it can only be accepted on behalf of the team that wrote and published the article.

Those most deserving of credit are the staff of *IEEE Micro*. We can safely say that without the effort by this group our article would not have been noteworthy at all. It is not at all difficult to recognize the particularly important efforts of Dr. Peter Rony, then Editor-in-Chief of *IEEE Micro* and Joe Schallan, the best copy editor with whom we have had the pleasure of working. These people and their staff made our article into what it is. And if we are fortunate enough to be recognized at all for this article, then we can accept that honor only on the condition that the other major contributors not named also be recognized and honored. While we cannot accept this award on our behalf alone, it is a pleasure to accept it on behalf of all the people that have worked so hard to make this article and *IEEE Micro* successful. We are the lucky beneficiaries of their efforts and we thank them.

Doug MacGregor
David Mothersole
Motorola, Inc.

CRC-16 flies better in assembler

To the Editor:

The April 1985 issue of *IEEE Micro* (pp. 67-75) has an interesting article by D.V. Shouse entitled, " 'On the fly' CRC-16 Byte-wise Calculation for 8088-based Computers." I don't understand however, why the author chose Fortran as the programming language, especially for the 8086/88 microprocessor. Fortran is really not optimized for bit manipulations.

Going back to A. Perez' article in the June 1983 issue of *IEEE Micro* (pp. 40-50), and with a little modulo 2 (exclusif OR) arithmetic, it can be shown that the CRC-16 (X**16+X**15+X**2+1) polynomial can be created by the modulo 2 addition of the four words listed in Figure 1. The body of an assembler program for the 8088 executing these calculations would appear as in Figure 2.

continued on page 99

MSbit															LSbit
0	0	0	0	0	0	0	0	C16	C15	C14	C13	C12	C11	C10	C9
Px	Px	0	0	0	0	0	0	0	0	0	0	0	0	0	Px
0	0	X8	X7	X6	X5	X4	X3	X2	X1	0	0	0	0	0	0
0	X8	X7	X6	X5	X4	X3	X2	X1	0	0	0	0	0	0	0

Notes:

- C*i* are the individual bits of the old CRC word value.
- X*i* are the individual bits of the modulo 2 addition of the data byte and the low byte of the old CRC.
- Px is the parity of the X byte.

Figure 1. The CRC-16 polynomial is created by the modulo 2 addition of four words.

```
3       32 DB           XOR     BL,BL       ;clear for even parity
3       32 D0           XOR     DL,AL       ;form X
4:16    7A 02           JPE     SKIP        ;skip if parity even
4       B3 07           MOV     BL,07       ;set for odd parity
                SKIP
2       8A FA           MOV     BH,DL       ;form third word
2       D1 CB           ROR     BX
3       32 FA           XOR     BH,DL       ;form last word
2       D1 CB           ROR     BX
3       32 DE           XOR     BL,DH       ;form new CRC
2       8B D3           MOV     DX,BX       ;save new CRC
-------
32 cycles mean

Input:   data byte in AL, old CRC in DX.
Output:  new CRC in DX and BX, AL is unchanged and the old value of BX is lost.
```

Figure 2. Assembler program to execute the CRC-16 calculation with the 8088 microprocessor.

FERMTOR: A Tunable Multiprocessor Architecture

Jonathan Rose, Wayne Loucks, and Zvonko Vranesic
University of Toronto

Multiprocessing can be cost-effective when a general-purpose system is adaptable to specific uses.

In the search for faster and more powerful computers, researchers have followed two paths. The first concentrates on increasing the speed of a uniprocessor. This can be achieved by making the components faster,[1] by using pipelining,[2] and by exploiting architectural features such as a cache memory and reduced instruction sets.[3]

The second approach is directed to gaining high performance through the use of more than one processor. Indeed, multiprocessors have often been considered a panacea for computing problems. Recent developments in Very Large Scale Integration (VLSI) technology have further motivated this work, because integration promises to make multiprocessing cheaper. This is manifested in three ways:

1) Systolic architectures place many small asynchronous processors in a regular array that can be implemented on one chip.[4] More conventional SIMD (Single Instruction stream, Multiple Data stream) architectures can also be highly integrated.
2) Microprocessors become more powerful as higher levels of integration allow the inclusion of more architectural features on a chip. Today's microprocessors are architecturally similar to yesterday's mainframes. Thus a multiprocessor architecture incorporating general-purpose microprocessors naturally becomes more powerful as technology improves.
3) The MIMD (Multiple Instruction stream, Multiple Data Stream) hardware for communication between processors can be integrated. A circuit that formerly required many TTL chips can be realized on one large-scale chip, limited principally by pin count. Thus, although data paths may need to be external to a VLSI chip, the communication protocol implementation and controlling logic can be integrated easily.

The ideal objective of multiprocessor structures in general, and MIMD architectures in particular, is to obtain linearly increasing throughput, dependent upon the number of processors. Rarely, however, will n processors be n times faster than one processor unless the application lends itself to being subdivided into many parallel subtasks. It is true that while two processors can be made to work almost twice as fast as one processor, this property does not hold for more general cases.

0272-1732/85/0800-0005$01.00 © 1985 IEEE

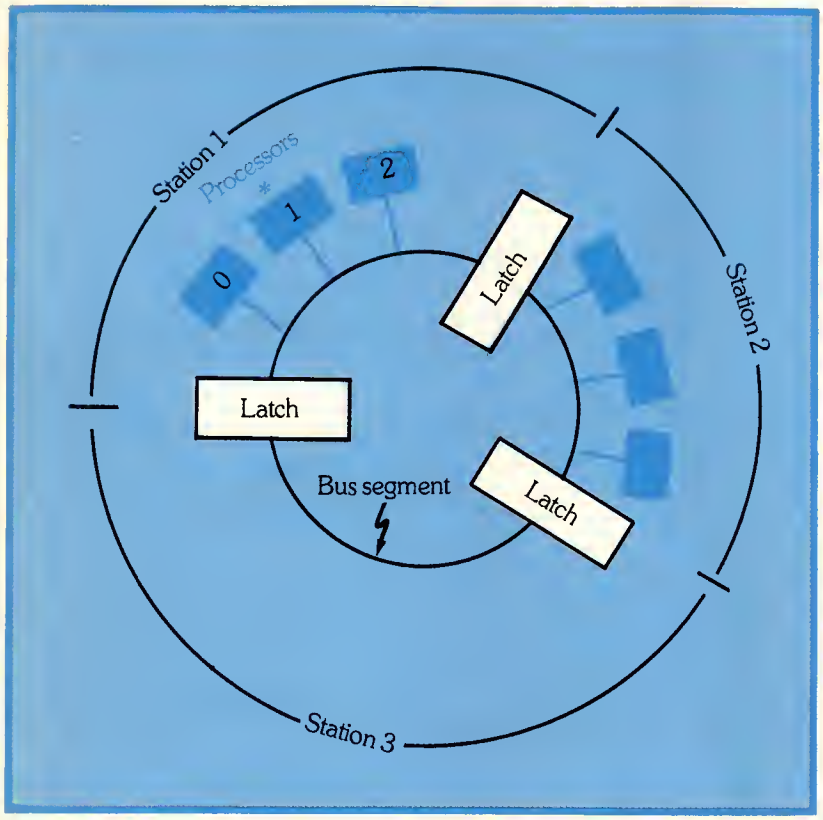


Figure 1. The basic FERMTOR architecture.

The greatest degree of success in multiprocessor systems occurs when they are applied to a specific purpose and the entire machine (i.e., hardware and software) is designed toward that purpose. SIMD processors such as ILLIAC IV[5] have been used for weather prediction, and architectures such as the Cytocomputer[6] have been used for image processing. Associative processors have been used in database machines.[7] MIMD computers have been successful when each processor is assigned a permanent task; for example, in recent personal workstations one processor handles the graphics screen, another acts as a disk server, and a third performs the computation.

However, it is very tedious to have to design and optimize a new architecture and communication system every time one requires a new MIMD system. It would be far more efficient if there were general architectures available, with a standard interprocessor communication scheme and programming environment that could be *tuned* to each new application. Such a system would allow fast design and construction of a powerful special-purpose multiprocessor. It could take advantage of an existing family of VLSI microprocessor chips intended for a general-purpose architecture.

Nevertheless, there is certain to be a trade-off between the efficiency of each implementation and the generality of the basic architecture. The tuned architecture *must* be cost-effective. If the architecture allows cost-performance trade-offs, it will be that much more valuable. An example of an interesting cost-performance trade-off is the SIMD Cytocomputer.[6] Rather than implement a full array of processors, this architecture pipelines pieces of the array through a sub-array processor.

Finally, one of the greatest difficulties in using the parallelism available in an MIMD system is the task of scheduling the work of each processor. The programming environment of a multiprocessor must address this problem directly.

In this article we present FERMTOR, an MIMD architecture developed at the University of Toronto with the goal of addressing the issues raised above.[8, 9, 10] The name stands for "Flexible Extendible Range Multiprocessor at TORonto." FERMTOR is a general MIMD architecture that can be tuned to many applications. It is also a practical multiprocessor whose communications hardware is much less complex than that of an MIMD crossbar or a Banyan network. It has packet-switched, combined-ring, and shared common-access bus interconnection schemes. Processors communicate directly with each other by means of packets. Each processor can be used for either general purposes or special purposes, such as for high-speed numerical computation. A simple version of FERMTOR has been constructed and tested.

FERMTOR has several features in common with a number of previous architectures, although we believe that its overall design provides certain unique characteristics. The manner in which packets flow around the ring, and the fact that a packet arrival actively interrupts a processor, likens FERMTOR to data-flow machines.[11, 12, 13] Farrel[14] uses a ring structure to implement a Generalized Control Flow (GCF) machine. The GCF architecture is a practical attempt at data-flow implementation. FERMTOR

also bears some similarity to CM*[15] in that *processes*, rather than lower level program units such as instructions, communicate among each other. Indeed, CM*, if configured as a packet-switching multiprocessor, could be used in a manner similar to FERMTOR.

Further, FERMTOR's ring and common-access bus structure resemble the topology of the EMMA architecture.[16] EMMA is used as a pattern-recognition machine for postal sorting and is highly successful, incorporating fault tolerance and graceful degradation under faults. The software structure of the prototype FERMTOR is also similar to HM2P[17] in that it uses Hoare's monitors and signals.[18] A great deal of other multiprocessing work exists, but little of it addresses the question of tuning to a specific purpose.

This article discusses the general architecture of FERMTOR, surveys the software structure of the programming environment, gives some details of the hardware implementation, and provides some results obtained with the prototype, including performance measurements. We also discuss two potential applications of FERMTOR and suggest avenues for future work.

FERMTOR architecture

FERMTOR is an MIMD architecture with a part ring, part shared common-access bus communication scheme. The basic structure is shown in Figure 1. The processors are connected to a parallel-pipelined bus called the *P-Bus*. The P-Bus is a ring-like structure of a number of bus segments. Parallel data flows synchronously between the latches, which delineate the bus segments. Each latch, bus segment, and group of processors following it is known collectively as a *station*. There can be a number of processors at each station, the number being limited by the desired bandwidth of the shared common-access bus. Four different types of processors are used:

1) General-purpose processors, such as conventional microprocessors.
2) Special-purpose hardware for high-speed computation, such as array processors.
3) Memory processors that contain and manage the global memory of the system.
4) Input/Output processors that control data flow between FERMTOR and I/O devices such as disks, terminals, etc.

Data is exchanged within a station using a shared common-access bus. Every processor on the P-Bus occupies a unique address by which it can be unambiguously referenced. The address consists of two parts: a station number and a processor number within that station. For example, the processor marked with an asterisk in Figure 1 is at station 1, unit number 1, and so has a P-Bus address of 11.

Basic packet structure. The "slot" of data within a station contains a *packet* of information. The packet is the basic unit of communication among all processors. A packet contains the following fields:

Source. The P-Bus address of the transmitting processor.
Destination. The P-Bus address of the receiving processor.
Operation. Specification of the nature of the packet.
Operand. Data to be operated on.

In addition, when slots travel between stations, three status bits are appended to specify the presence or absence of a packet in the slot and whether or not the packet has been successfully received by the destination station.

P-Bus operation. At each station, a *station manager* controls the flow of packets among local processors and between the neighboring stations. Figure 2 is a block diagram of one station and the P-Bus interface of two processors. When a processor wishes to transmit a packet, it raises a **Request** signal to the station manager. The manager arbitrates the processor requests and sends an **Enable** signal to the selected processor when the first empty packet arrives from the previous station. The requested transfer can be either a *local* transfer between processors in the same station or a *nonlocal* transfer between processors at different stations.

Local transfers. When the packet destination is local to the station, the manager tries to transmit it to that processor as soon as possible over the shared common-access bus. If the destination processor is unable to accept the packet (as

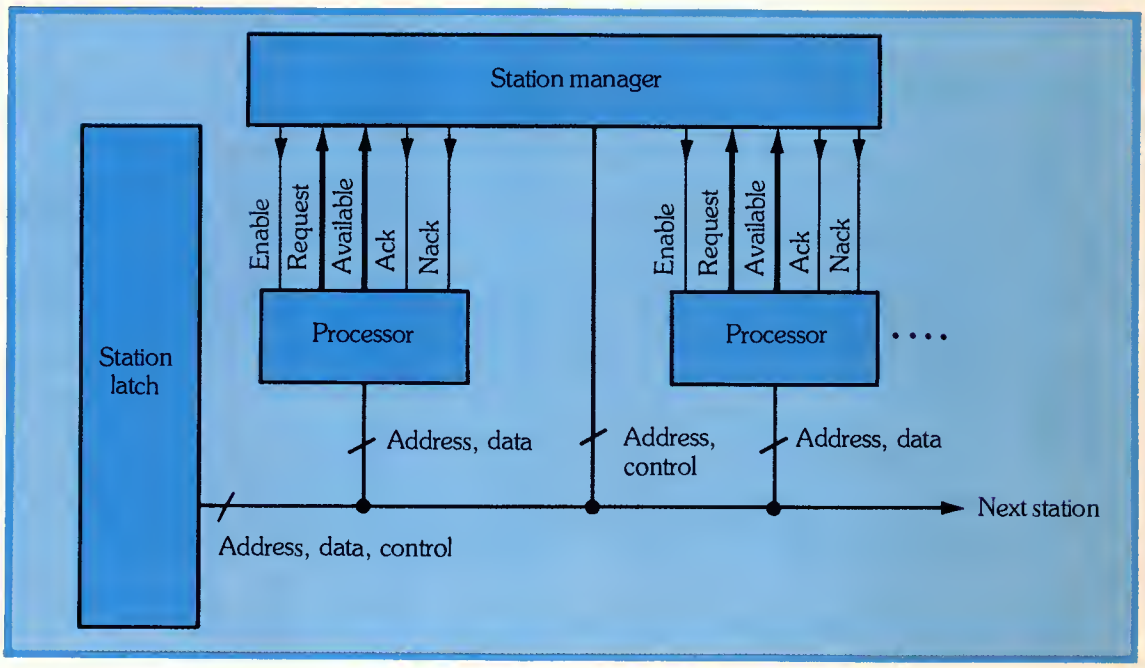


Figure 2. A typical FERMTOR station.

determined by its **Available** signal output) then a **Nack** signal (negative acknowledge) is sent to the transmitting processor. If the packet is accepted, then an **Ack** signal (acknowledge last transfer) is sent to the transmitting processor. The packet slot occupied by this transaction (whether successful or not) is then marked **Empty**. This means that the slot is immediately available for use by succeeding stations on the ring—i.e., a local transfer is always concluded in the time it takes one packet to go through a station.

Nonlocal transfers. When the packet destination is not local to the station, then the packet is transmitted to the next station along the ring. Each station that the packet passes through checks to see if it is the destination station. When the destination station is reached, it tries to transmit the packet to the destination processor exactly as described above for local transfers. If the transmission is successful, the packet is marked **Received**; if not, the packet is marked **Not Received**, using one of the appended flags. When the packet returns to the source station, the station manager sends either an **Ack** or a **Nack** signal to the source processor, depending on how the packet was marked.

Note that, from the processor's point of view, there is no difference between intrastation transfers and interstation transfers, other than the time it takes to transmit the packets. For any unsuccessful transfers, the processor simply regenerates a **Request** signal until the transfer succeeds. The P-Bus protocol allows a processor to have only one packet active on the ring at one time, to prevent it from dominating ring traffic.

Flexibility and extendibility of the architecture. The flexibility of the FERMTOR architecture facilitates the addition of stations and processors. When a processor becomes a bottleneck in a given computation, a second processor of this type can be added to take on a share of those computations, provided the problem is divisible. This is part of the process of tuning the FERMTOR architecture to a special-purpose application.

The P-Bus ring structure makes the hardware complexity of FERMTOR directly proportional to the number of processors. If a processor is added to an existing station, the increase in communication hardware is minimal. When a new station is added, the full station-manager hardware must be included.

The exact configuration of a FERMTOR implementation can be tuned to achieve sufficient interprocessor communication speed. There is a trade-off between having fewer processors at a

station and thus many stations, or many processors at a station and few stations. If there are only a few processors at a station, then they can communicate quickly among themselves. However, because there are many stations, the latency (the time required for a packet to traverse the ring) becomes larger. Conversely, with many processors per station and fewer stations the latency is small. In this case the intrastation bus contention is larger because many processors share one bus. Thus, the optimal number of processors per station is application-dependent. Loucks[9] found that three or four processors per station was best for general-purpose computation.

If the FERMTOR structure is viewed as a hierarchy, we can put the question of the number of processors per station in perspective. Further levels of hierarchy can be added to the P-Bus by using multiple rings, each communicating via inter-ring bridges. Thus an implementation could be a tree of rings or even a ring of rings. The farther the destination of a packet is from its source (in terms of levels of the hierarchy it must travel) the greater the time the packet transportation takes.

If it is apparent from the application that two or more processors communicate very frequently, then they should be placed at the same station. Groups of processors that communicate less frequently, but still at a significant rate, should be placed on the same ring. If two distinct groups of processors have little need to communicate, then they should be placed on separate rings. In this manner a FERMTOR configuration can partition the processors as required by the application.

The hierarchy of packet transportation is similar to CM*[15]; but, since it can be extended indefinitely, it is cleaner and more symmetrical.

Software environment

The viability of a multiprocessor is determined not only by its architecture, but by the software infrastructure as well. The communication scheme for the prototype FERMTOR allows versatile packet level communication between processors.

Granularity of parallelism. The use of the packet communication structure depends upon the choice of processor. Microprogrammed bit-slice processors[9] allow very fast interaction with the P-Bus. General-purpose microprocessors using memory-mapped I/O to access the P-Bus registers are significantly slower. Indeed, it requires roughly 12 microprocessor instructions (at 3 to 4 microseconds each) simply to load the P-Bus latches and request a transmission. It takes a great deal more time to determine the contents of that packet.

A microprogrammed machine could access the P-Bus on an instruction basis—i.e., every machine-level instruction would use the P-Bus. But this is not possible for the slower microprocessor elements. In this case we must ensure that each processor uses the P-Bus less frequently. Thus for such microprocessors, the basic unit of parallelism is the *process* and not the instruction. The process is constrained to executing local object code and performing intraprocessor communication at a rate compatible with the P-Bus interface. Later in the article we discuss how to relax these constraints by means of a faster interface.

FERMTOR uses processes as its basic unit of parallelism. The coarse granularity of this parallelism increases the likelihood that a processor is left idle for a long period of time while waiting for another process to finish a calculation. This could happen, for example, when one processor requests data from another that requires significant computation—the requesting processor would be idle during the computation. To make use of this idle time, each processor in FERMTOR is itself multiprocessing. That is, every processor can have several active processes in it so that it is busy as much as possible. While one process awaits data from another processor, other processes can use the microprocessor. The implementation of multiprocessing was done using the concurrent programming language Concurrent Euclid.[19] It is a dialect of Pascal and provides multiprocess synchronization using Hoare's monitors and signals.[18] Note that the programs are compiled individually for each processor, not as one big program for the entire multiprocessor.

Implementing software in a concurrent language has another benefit: the interprocessor communication can be run by separate processes of which the user need have no knowledge.

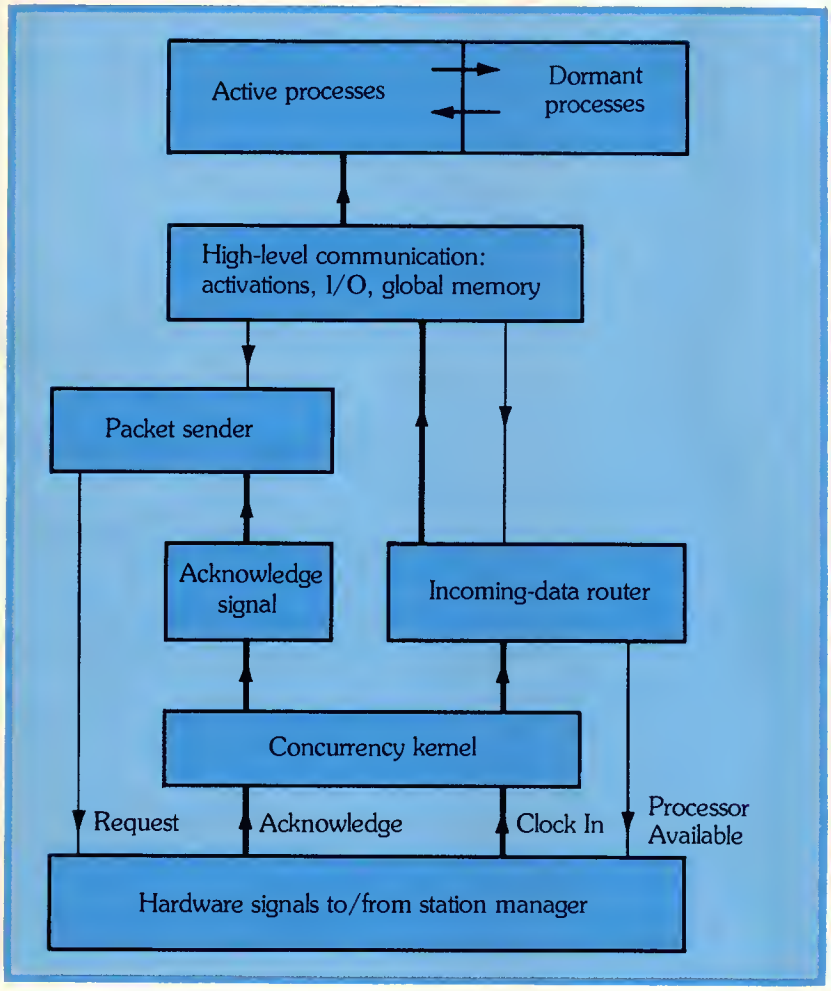


Figure 3. The communication hierarchy along the P-Bus.

Coarse granularity. The decision to choose the process as the granularity of parallelism is a significant one. The choice was between a fine granularity such as instruction-wise parallelism and a coarse granularity such as parallel processes.

Instruction-wise parallelism occurs when a very small amount of computation is done between interprocessor transfers. For example, this could be the amount of computation involved in a typical high-level-language arithmetic statement. In this kind of parallelism, each processor is explicitly told to execute such a statement by a separate scheduling processor. The scheduler communicates with the executing processor over the communication bus. Many statements are scheduled into several processors and ordered into a "chain" of instructions to be sequentially executed. The first processor/instruction is activated, then executed, and that processor activates the second instruction in the chain, and so on.

Instruction-wise parallelism incurs a great deal of overhead per instruction. Loucks[9] investigated this and found scheduling to be a significant bottleneck. Scheduling will always be a problem if the amount of computation per interprocessor transfer bus is small.

Process-wise parallelism occurs when a large amount of computation is performed between interprocessor transfers, such as would be needed to solve a large system of linear equations, for example. Typically, one processor acts as a scheduler, assigning these large pieces of the overall task to slave processors.

Parallel processes greatly reduce the P-Bus traffic because the processors spend much more time calculating between bus transfers than they do with instruction-wise parallelism. In addition, process-wise parallelism makes it relatively more easy to schedule large chunks of processor time.

In FERMTOR, we chose process-wise parallelism. The issue of scheduling is discussed further below.

Note that parallelism granularity is a continuum rather than a discrete choice between coarse and fine. It is an open problem to discover exactly how much computation should be done between interprocessor transfers. The answer is most likely both application- and situation-dependent.

Communication hierarchy. Processes can communicate at many levels. They can simply send a byte of data or transfer an entire file. They can also send messages which control program flow directly. To support these levels of communication there must exist a coherent substructure of software. This idea is not unlike the protocol levels described in the ISO model for Open System Interconnection,[20] commonly used in local-area networks. In FERMTOR, the coupling between processors is much greater than with a LAN, but similar ideas still apply. Figure 3 is a block diagram of the communication hierarchy.

Low-level packet communication. At the lowest level in the hierarchy is the actual hardware that performs the physical data transfer

and handles signalling between the processor and station manager. Here a packet is the basic unit of transaction. The first level of software interface is the *concurrency kernel*. The kernel, which is part of the Concurrent Euclid programming language, implements concurrency within the microprocessor. This includes handling hardware interrupts from the processor's P-Bus interface. The kernel polls the station manager after an interrupt to determine the cause, and then dispatches the associated Euclid process.

The **Ack** (acknowledge) signal from the station manager, shown in Figure 3, indicates that the previously requested P-Bus transfer has been completed. This interrupt is handled by the kernel, which invokes the acknowledge process. The acknowledge process sends a software signal[18] to the *packet sender*, which is then free to initiate another transmission. The packet sender raises the hardware request signal to transmit a packet.

The **Clock In** signal from the station manager indicates that a new packet has arrived for the processor. Again, this interrupt is handled by the kernel, which then invokes the *incoming-data router* process. Any process that expects to receive a packet must inform the incoming-data router of the type of packet it expects.

The incoming-data router keeps a table of all the packet types in the processor, along with the associated processes. When the router receives a packet, it signals the associated process and gives it the packet's contents.

High-level packet communication. In addition to low-level communication, Figure 3, described above, also depicts three kinds of medium- to high-level communication:

1) *Data input/output.* One or more processors are usually designated as the I/O processors, and are dedicated to that purpose. All other processors transmit data (for output) and request data (for input) from the I/O processors.

2) *Global memory.* One or more processors are designated to act as global memory. The memory processor performs four functions:
   - Allocate a block of storage.
   - Deallocate a block of storage.
   - Write to global memory.
   - Read from global memory. The read/write functions are similar to the I/O functions.

3) *Activations.* A process in the processor can be dormant waiting for another calculation to finish. An activation packet will wake that process and tell it where to find the data that it is waiting for; often such data is stored in global memory. Note that this feature has overtones of the data-flow concept.

Scheduling. Scheduling the processing resources of a multiprocessor is a crucial and difficult task. In the prototype FERMTOR, the process itself must be scheduled. This is the task of deciding which process goes to which processor. The schedule is currently static, but could be made dynamic if an efficient method of transporting code were developed (see our discussion of enhancements below). In the present version of FERMTOR, there is no automatic scheduling. The user partitions the processes, attempting to get maximum performance from the processors.

Using Concurrent Euclid, it is a simple matter to determine how much time each processor is idle. This knowledge can help the user to partition a multiprocessor program. Note that the work required to produce a good partition is only justified if the application is going to be used over a long period of time. This style of scheduling is applicable to special-purpose machines that have processors dedicated to permanent tasks.

Implementation

We have constructed a prototype FERMTOR that is sufficiently large to contain all the salient features of the architecture. It has provided a test vehicle to assess the viability of the architecture.

The prototype consists of three stations that are capable of supporting four processors each. We currently have six processors, which can be distributed arbitrarily among stations. Five of the processors are Motorola 6809 general-purpose microprocessors with their own local memory and P-Bus interface hardware. The sixth processor is a PDP-11/34 running the UNIX operating system. A special P-Bus interface to the PDP-11 was constructed, using a standard parallel port on the UNIBUS.

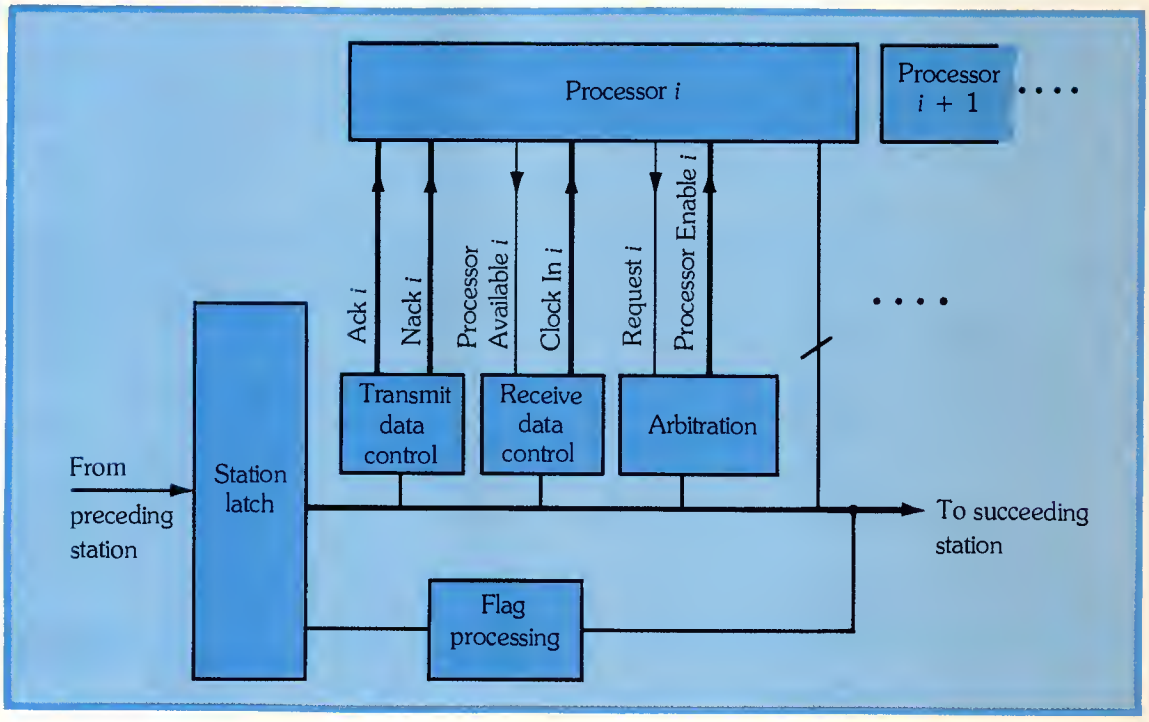


Figure 4. A detailed view of a station manager.

Station manager. A block diagram of the prototype station manager is shown in Figure 4. It consists of four principal parts. The *station latch* holds the data packets transmitted between the stations. The *arbitration unit* decides, every cycle, which packet is enabled onto the local bus. If the incoming ring packet is full, then it takes precedence. If it is empty, then the arbitration unit arbitrates processor requests for the bus. The *receive data control unit* transmits any packets destined for this station to the addressed processor by clocking the bus data into the processor's buffer. It also sends the **Clock In** interrupt signal to the processor. The *transmit data control unit* determines the success or failure of transmissions from this station and informs the sending processor with an **Ack** interrupt or a **Nack** signal.

The P-Bus is synchronously clocked. The maximum speed is determined by the arbitration unit and is roughly 2.5 MHz. A two-phase clock is generated centrally and is distributed to all the station managers. Other timing signals are generated local to each station from this clock.

Processors. Figure 5 is a block diagram of the 6809 processor and its P-Bus interface. The P-Bus interface has two parts: the buffers that contain incoming and outgoing data from the station bus, and the control circuit that handles the following control signals.

- **Request:** an output request to the station manager, indicating that there is data in the output buffer to be transmitted.
- **Processor Available:** an output-status flag indicating that the processor is able to receive another packet. This circuit is made difficult because the processor can be either much slower or much faster than the station manager.
- **Processor Enable:** an input from the station manager that causes the processor to place a requested transmission onto the P-Bus.
- **Clock In:** an input from the station manager that sends a packet from the P-Bus to the processor. This signal interrupts the processor.
- **Acknowledge:** an input from the station manager indicating that the last request was successful. It is wired to cause an interrupt to the processor.
- **No Acknowledge:** an input from the station manager indicating that the last re-

quest was not successful. Currently, this signal is wired to generate a new request. It could also be wired to input directly into the processor so that multiple transmission failures could be detected, and ring faults deduced.

Results and performance

The prototype FERMTOR consists of three stations and six processors. Each station Manager fits on one wire-wrapped board. One 6809 processor is a wire-wrapped prototype, and the other four are implemented as printed circuit boards. There is also an interface board that connects a PDP-11/34 with the P-Bus.

The P-Bus and associated processors have worked error-free in all tests. We have exercised the multiprocessor in a number of ways, testing communication speed alone as well as its multiprocessing capabilities.

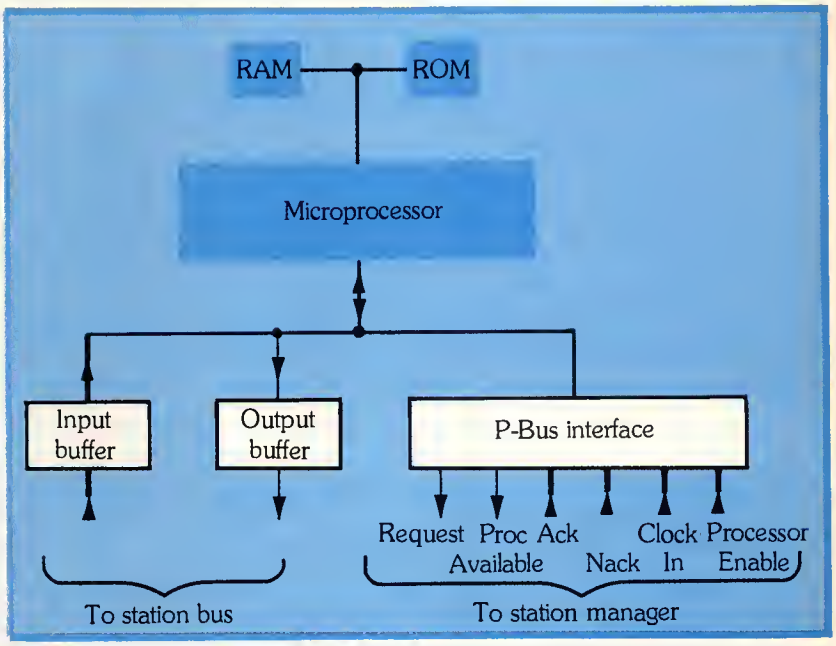


Figure 5. A detailed processor diagram.

P-Bus speed test program. The station manager clock on the prototype FERMTOR is set at 1 MHz. The effective bit-transfer rate of the P-Bus, which carries 24 bits of data, is 24M bits per second. The hardware is capable of supporting a 2.5 MHz clock, and so the maximum P-Bus transfer rate is 60M bits per second. This is the maximum amount of data that can be transmitted around the ring by all of the processors. A higher transfer rate can be achieved if there is a significant amount of intrastation transfers. Loucks [9] gives an analysis of the effective P-Bus transfer rates.

Standard microprocessors that use memory mapped I/O require only a small fraction of this bandwidth, as discussed immediately below. Therefore, the P-Bus is capable of supporting a very large number of such processors, in the range of several hundred per ring.

Transfer rates per microprocessor. P-Bus input/output by the microprocessor is done using latches that are mapped into processor memory. Every P-Bus transfer requires the loading and unloading of these latches, at the same rate as a load/store memory access.

We measured the maximum number of transfers that the microprocessor can perform. For both transmission and reception, the maximum rate is roughly 4000 packets per second. Since the P-Bus can support 2.5 million transfers per second, this suggests that several hundred processors can be supported on a one ring P-Bus.

If a higher per-processor transfer rate is required, the enhancements discussed further below could be implemented.

A simple test. To test the basic multiprocessing capability of FERMTOR, we wrote a simple application program. The program tests a range of integers for primeness. In each slave processor, a program waits for two packets specifying a number range to be tested. It uses the simple algorithm of dividing the number under test by all the odd numbers up to one third of that number and checking for a remainder. This inefficient algorithm makes the calculation highly CPU-intensive. The master (or scheduling) processor sends out the number ranges to the slaves. When a slave determines that a certain number is prime, it transmits that number to the master.

The large amount of calculation required by the test makes P-Bus transfers very infrequent. As a result, FERMTOR exhibits an ideal multiprocessing property: the computation speed is linear with respect to the number of processors.

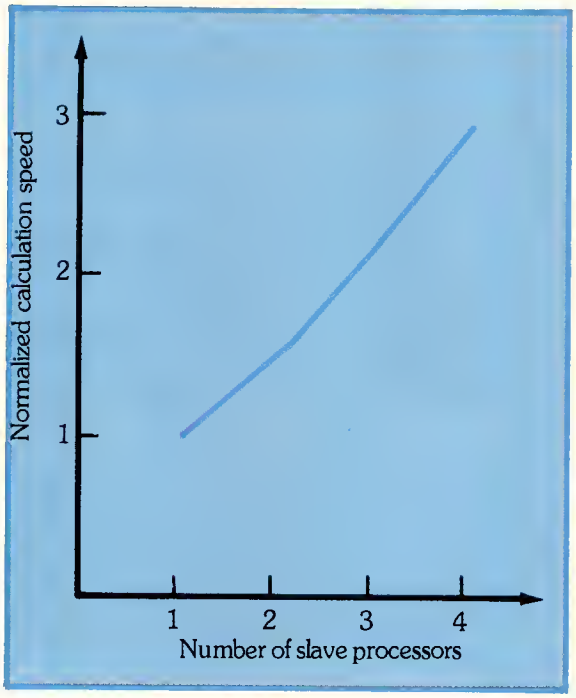


Figure 6. Normalized calculation speed versus number of slave processors. The linear arc of the curve indicates ideal multiprocessor operation at up to four slave processors.

That is, if for two processors (one master and one slave) the time taken is t, then for three, four or five processors it is $t/2$, $t/3$ and $t/4$, respectively. This performance is due entirely to the fact that the processors are loosely coupled.

A closely coupled multiprocessing test. It is generally true that the more closely coupled a multiprocessor architecture allows its processors to be, the more applicable that multiprocessor structure will be. Thus we wished to test FERMTOR with a multiprocessor program that frequently uses the P-Bus.

We again chose the prime number example but adapted the algorithm for more cooperation between processors. We wished to determine all the primes between 1 and some number X. As the system discovers these primes, at first using the same algorithm as above, the master processor records them and distributes them back to the slave processors. Thus the slaves need only divide the number under test by the prime numbers less than one third of the number. Note that if there are not sufficient prime numbers computed to that point, this algorithm again reverts to the former one. The transmission of primes to the master and their subsequent rebroadcast places a much heavier load on the P-Bus than does the simple test.

The size of the number range that each processor tests at a time becomes a factor in this algorithm. The smaller the range, the more frequently the master must present a new range, but the larger the range, the longer it takes to transmit discovered primes to all the slaves.

The results obtained with this closely coupled example are shown in Figure 6. The figure is a plot of normalized calculation speed versus the number of slave processors. The normalized calculation speed for n slave processors is found by dividing the calculation time of one slave processor (for the entire task) by the time required for n processors to do the entire task. Ideally, the normalized calculation speed increases linearly with the number of processors. The curve in Figure 6, which is indeed linear, demonstrates that we achieved the ideal multiprocessor result for up to four processors. The difference between this example and the less closely coupled example above is that the slope of the line is less than 1. For this example the slope is 0.64. This is due to the interprocessor communication overhead. There is increased interprocessor communication when more than one slave processor is calculating. This overhead is constant per processor, as indicated by the linear curve in Figure 6.

It is interesting to note how the interprocessor communication was optimized in this example. By measuring how much each processor was idle, and determining why it was idle, it was possible for us to discover bottlenecks in the multiprocessor program. For example, we found that the packet transmission by the scheduler was a bottleneck. Instead of having the scheduler transmit discovered primes to all the slaves, we altered it to transmit to only one slave. That slave then forwards the number to the next slave and so on until all slaves receive the prime number. Thus in the four-slave case, the scheduler need only send out one packet per prime instead of four. This eliminates the packet transmission bottleneck in the scheduler.

This kind of optimization produces a constant interprocessor-communication overhead per processor. The technique of idle-time measurement tells not only how idle each processor is, but exactly why it is idle. Provided a task can be divided

into many parts, then this idle-time analysis permits *fine tuning* to get good utilization of the multiprocessor.

Applications

We propose FERMTOR as a multiprocessor architecture tunable to other special-purpose applications. We are investigating three such applications: simulation, VLSI layout, and signal processing.

Simulation. Computer simulation is a task well-suited to the application of parallelism. Many physical systems that are simulated by computer are inherently parallel. At the same time, simulation may require many computation hours to complete, so that parallelism can reduce this considerably. Intuitively, it appears that FERMTOR would lend itself to implementing a language like SIMULA.[21] Some work has already been done on using MIMD systems and SIMULA.[22] In FERMTOR, the flow of data (such as packets around a ring) and control (such as activations) is very similar to the communication between SIMULA classes.

We are developing a few simple constructs to add to Concurrent Euclid to aid in simulation. Processes are activated by packets containing data, upon which the process operates. There are two kinds of processes:

- *Nonqueued processes,* in which any number of activations can be active at a given time (using reentrant code); and
- *Queued processes* (representing a single shared resource), in which only one activation is active at a given time. Subsequent activations are placed in a FIFO queue.

Perhaps the most difficult problem in parallel simulation is that of simulation-time synchronization. Every processor must agree that a certain piece of data or state is associated with the same simulation time. This may mean that all processors must wait for the slowest process to be completed before the next time period begins. An alternative is the use of *time stamps.*[23, 24] Here any message of data or activation contains a stamp indicating the time at which the sending processor is operating. The receiving processor cannot use the message until it reaches that time. Some synchronization problems remain; these are discussed by Reynolds.[24] The work on parallel simulation is in its infancy and requires a great deal of work on compiler generation and the issue of synchronization.

VLSI layout. The automatic layout of VLSI circuits, already a time-consuming task, promises to grow evermore compute-bound as the density of VLSI circuits increases. The partitioning, placement, and routing of VLSI circuits is a prime candidate for the application of multiprocessing. Much of the existing work concentrates on the use of SIMD structures for placement and routing.[25, 26, 27] We intend to use MIMD structures for dealing with layout problems. MIMD structures can make better use of existing layout algorithms than SIMD architectures because of the greater similarity between MIMD multiprocessors and uniprocessors. For example, the natural hierarchy of VLSI circuits can be used as a technique for partitioning the circuit, so that individual processors can do traditional placement on a small section of the circuit. We are also investigating the implementation of an algorithm like Soukup's Global Router.[28] Here one processor would be responsible for one interconnection net, giving the potential of a large amount of parallelism.

Signal processing and synthesis. The prototype P-Bus is currently being used to connect a number of Motorola 6809 and TMS 320 processor boards for the purposes of both signal processing and signal synthesis.[29] The TMS 320 is a high-speed, limited-memory processor. Several TMS 320s are being used with the P-Bus to cascade high-speed calculations for good-quality digital filters and for music synthesis. The versatility and tunability of the P-Bus permits easy addition of the TMS 320 processors.

Enhancements

We are currently working on several aspects of FERMTOR. It is clear that, using process-wise parallelism, faster individual processors will speed up the multiprocessor in a cost-effective way. We are now implementing a FERMTOR with National 32016 system microprocessors.[30]

To speed up the interprocessor communication, we are using DMA to implement two features:

- *Implicit shared memory*. Part of the local processor's memory will be mapped to another processor's memory, and any access to that memory will automatically use the P-Bus to get the remote data. This will eliminate explicit processor transfer requests.
- *Block transfer requests*. Hardware will use DMA to transfer large blocks of data and code between processors very quickly. Here the processor will explicitly initiate the request but will not be involved in each individual packet transfer. This is similar to DMA transfer between magnetic disks and processors.

These new features will be added to the 32016-based processors through the MULTIBUS on each processor.

Future application development work will concentrate on the VLSI layout machine. This promises to be an extremely useful application of FERMTOR.

We have presented a multiprocessor architecture that can be easily tuned to fit many applications. Through our experiments on the prototype, we found the potential for achieving a great degree of parallelism, provided a task is well partitioned. We are continuing to develop applications such as simulation and VLSI layout. The FERMTOR architecture provides a viable structure for developing powerful machines using standard microprocessor components.

References

1. H. Morkoc and P.M. Solomon, "The HEMT, A Superfast Transistor," *IEEE Spectrum,* Vol. 21, No. 2, Feb. 1984, pp. 28-35.
2. J.H. Hennessey, et al., "Design of a High Performance VLSI Processor," *Proc. VLSI 83,* pp. 1-21.
3. D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer,* Vol. 15, No. 9, Sept. 1982, pp. 8-21.
4. H.T. Kung, "Why Systolic Architectures?" *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 37-46.
5. G.M. Barnes, et al., "The ILLIAC IV Computer," *IEEE Trans. Computers,* Vol. C-17, No. 8, Aug. 1968, pp. 746-757.
6. R.A. Lougheed and D.L. McCubbrey, "The Cytocomputer: A Practical Pipelined Image Processor," *Proc. 7th Ann. Symp. Computer Architecture,* May 1980, pp. 271-277.
7. P.B. Berra and E. Oliver, "The Role of Associative Array Processors In Data Base Machine Architectures," *Computer,* Vol. 12, No. 3, March 1979, pp. 53-61.
8. W.M. Loucks and Z.G. Vranesic, "FERMTOR: A Flexible Extendible Range Multiprocessor," *Proc. Canadian Information Processing Soc. Conf.,* 1980, pp. 134-145.
9. W.M. Loucks, "FERMTOR: A Flexible Extendible Range Multiprocessor," PhD thesis, University of Toronto, 1980.
10. J.S. Rose, "An Implementation of FERMTOR: A Flexible Extendible Range Multiprocessor," MASc thesis, University of Toronto, 1982.
11. J. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Trans. Computers,* Vol. C-26, No. 2, Feb. 1977, pp. 138-146.
12. J.B. Dennis, "The Varieties Of Data Flow Computers," *Proc. 1st Int'l Conf. Distributed Computing Systems,* Oct. 1979, pp. 430-439.
13. I. Watson and J. Gurd, "A Practical Data-Flow Computer," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 51-57.
14. E.P. Farrel, N. Ghani, and P.C. Treleaven, "A Concurrent Computer Architecture and a Ring Based Implementation," *Proc. 6th Ann. Symp. Computer Architecture,* April 1979, pp. 1-10.
15. R.J. Swann, S.H. Fuller, and D.P. Siewiorek, "CM*—A Modular Multiprocessor," *AFIPS Conf. Proc.,* Vol. 46, 1977, pp. 637-644.
16. L. Stringa, "EMMA: An Industrial Experience On Large Multiprocessing Architectures," *Proc. 10th Symp. Computer Architecture,* June 1983, pp. 326-333.
17. K.G. Shin, Y-H Lee, and J. Sasidhar, "Design of HM2P—A Hierarchical Multimicroprocessor for General Applications," *IEEE Trans. Computers,* Vol. C-31, No. 11, Nov. 1982, pp. 1045-1053.
18. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM,* Vol. 21, No. 8, Aug. 1978, pp. 666-677.
19. R.C. Holt and J.R. Cordy, *Specification of Concurrent Euclid,* Computer Systems Research Group Technical Report CSRG-133, University of Toronto, Aug. 1981.
20. H. Zimmerman, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Communications,* Vol. COM-28, April 1980, p. 425-432.

21. G. G. Birtwistle, L. Enderin, M. Ohlin, and J. Palme, *DEC System 10 SIMULA Language Handbook, Part I,* Swedish National Defence Research Institute and Norwegian Computing Centre, Stockholm, Sweden, 1978.

22. P. Georgiadis, M.P. Papazoglow, and D.G. Maritsas, "Towards a Parallel SIMULA Machine," *Proc. 8th Ann. Symp. Computer Architecture,* May 1981, pp. 263-278.

23. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* Vol. 21, No. 77, July 1978, pp. 558-565.

24. P.F. Reynolds, "A Shared Resource Algorithm for Distributed Simulation," *Proc. 9th Ann. Symp. Computer Architecture,* April 1982.

25. M.A. Breuer and K. Shamsa, "A Hardware Router," *J. Digital Systems,* Vol. IV, Issue 4, 1981, pp. 393-408.

26. R.A. Rutenbar, T.N. Mudge, and D.E. Atkins, "A Class of Cellular Architectures to Support Physical Design Automation," *IEEE Trans. Computer-Aided Design,* Vol. CAD-3, No. 4, Oct. 1984, pp. 264-278.

27. K. Ueda, T. Komatsubara, and T. Hosaka, "A Parallel Processing Approach For Logic Module Placement," *IEEE Trans. Computer-Aided Design,* Vol. CAD-2, No. 1, Jan. 1983, pp. 39-47.

28. J. Soukup, "Global Router," *Proc. 16th Design Automation Conference,* June 1979, pp. 481-484.

29. J. Kitamura, MASc thesis, University of Toronto, in progress.

30. *NS32000 Data Book,* National Semiconductor Corporation, 2900 Semiconductor Drive, Santa Clara, CA 95051.

Jonathan Rose is currently working on his PhD in electrical engineering at the University of Toronto, Ottawa, Canada. His research interests include multiprocessing hardware and operating systems and their application to automatic VLSI layout. During the summer of 1983, Rose was with Bell-Northern Research, Ltd, Ottawa, in the Integrated Circuits CAD/CAM group.

Rose received the BASc in engineering science in 1980 and the MASc in electrical engineering in 1982 from the University of Toronto. A junior fellow of Massey College of the University of Toronto, he is a member of the IEEE and the ACM.

Wayne Loucks is an assistant professor in the Department of Electrical Engineering at the University of Toronto. Prior to joining the university staff, he was a research associate involved in the development of a local-area computer network. His research interests are in computer architecture, multiprocessors, processing arrays, and LANs.

Loucks received the BASc in 1975 from the University of Waterloo, Ontario, and the MASc and PhD degrees from the University of Toronto in 1977 and 1980, respectively. He is a member of the IEEE and the ACM.

Zvonko Vranesic received the BASc, MASc, and PhD degrees from the University of Toronto in 1963, 1966, and 1968, respectively. In 1968 he joined the faculty of the Departments Electrical Engineering and Computer Science at the University of Toronto. During academic year 1984-85 he was on research leave at the Institut de Programmation, Universite de Paris 6.

Vranesic's research interests include computer architecture, multiprocessor systems, fault-tolerant computing, local-area networks, and many-valued switching systems. He has served as chairman and technical program chairman for the International Symposium on Multiple-Valued Logic. Vranesic is a senior member of the IEEE.

Questions about this article may be addressed to the authors at the Department of Electrical Engineering, University of Toronto, Toronto, Canada M5S 1A4.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 150 Medium 151 Low 152

FEATURE

An Interactive Debugging Environment

Frits van der Linden
Philips Medical Systems

Ian Wilson
BSO/Automation Technology

A Forth-inspired debug language, Fifth, provides an environment in which testing and debugging time is signifcantly reduced.

Debugging is the process of detecting and eliminating program errors. Static debugging involves freezing the system at some point in time and examining the system state—that is, the data structures and I/O ports. Dynamic debugging is performed concurrently with the execution of the user program and involves examination of the system state under conditions of near-normal system activity. This is normally achieved by freezing the system momentarily and taking a "snapshot" of the object of interest.

Real-time control programs are usually written as a collection of loosely coupled, concurrent tasks under the control of a real-time executive or operating system. Synchronization of and communication between tasks is by means of semaphores and messages. Debugging a multitasking system is usually much more difficult than debugging a single-tasking or sequential program. The reason for this difficulty lies mainly in the asynchronous and concurrent nature of multitasking systems. For example, faults can be caused by improper synchronization, by race conditions, or by improperly controlled access to data shared between processes. The difficulty of debugging multiprocessor systems depends largely on the degree of coupling between the various processors.

Testing is the process of demonstrating that a program performs according to its specifications. Testing frequently reveals the presence of bugs. Using debug tools, one can determine the exact cause of a particular bug. Correcting a bug at the source level usually requires leaving the debugging environment, entering a programming environment, building a new program, and returning to the debugging environment. These actions can be very time-consuming even if the various environments are available on the same workstation. Consequently, a programmer needs to be able to find as many bugs as possible before leaving the debugging environment. To maximize the usefulness of the test session, the test system should allow the programmer to correct or "patch around" simple bugs.

0272-1732/85/0800-0018$01.00 © 1985 IEEE

Integration is the process of combining various components—both hardware and software—to build a working system or subsystem. Modern system development philosophy, with its strong emphasis on prototyping and early user involvement, leads to smaller and more frequent debug-integrate-test cycles than more traditional approaches. Systems constructed from well-tested components having had some degree of exposure to users can reasonably be expected to meet their users' real needs.

The efforts required to perform the "integrate and test" phase of the system life cycle has all too frequently been underestimated. Major sources of difficulty include

- dependencies between components, either explicit (due to poor design philosophy) or implicit (due to hidden dependencies such as accidental timing relationships between tasks),
- surprises at the hardware/software interface level caused by ambiguous specifications and/or the unavailability of a realistic test framework until late in the development, and
- inadequately tested components (due to attempting to integrate too much too late in the development, or to the lack of suitable test/debug tools).

We felt the need for an interactive debugging environment that offers the programmer a considerable degree of control over the system rather than just passive inspection of data structures. Equally important was that the debugging environment be programmable, offering powerful primitives with which to build tools suitable for the debugging job at hand.

The debugging environment which we have implemented and fully integrated into our development process is the subject of the remainder of this article. We discuss debugging tools, general requirements for an environment suitable for debugging, salient features of our approach, and some underlying implementation issues. We conclude by summarizing our experience.

Debugging tools

It is useful to classify debug tools into several categories according to their main area of application (note, however, that the usefulness of most real debug tools spans a range of levels:

- *Hardware levels.* This normally involves specialized equipment such as the logic analyzer or in-circuit emulator. Signal levels and transitions can be recorded, and the instruction stream of the processor can be stored in a trace buffer. This type of information may be vital for diagnosing bugs caused by time-critical conditions.
- *Instruction level ("static" debuggers).* Implemented as ROM-resident "monitor" programs or as special "debug" programs, these debuggers provide display and modification of registers, memory contents, and I/O ports, and they allow breakpoint setting on execution of particular instructions, single stepping, and program disassembly. Bug fixes or patches are made using physical addresses and in hex or octal representation.
- *Operating system level ("static" or "dynamic" debuggers).* These debuggers access and display operating system data structures (for instance, the ready list or task control blocks) and usually provide breakpoints on system-related actions such as task activation. Task synchronization and communication objects are accessible. System and user entities can be referred to symbolically.
- *High-level-language statement level.* Information is presented in terms of the source-code statement that is being executed (as opposed to the object-code orientation of instruction-level debuggers). Program variables can be displayed and changed. Source-statement-level patching is available in interpreted languages such as APL and Interlisp, and under PSCOPE.[1]
- *Application level.* Information is presented in the vocabulary of the application rather than in terms of the programming language or operating system. User participation is usually

restricted to modification of selected system parameters. These debug tools are typically created on an ad hoc basis during the development process.

The implementation of medium to large real-time systems can be viewed as a sequence of integration phases. Each such phase has particular requirements that may be best satisfied by specialized debug tools:

• *Hardware integration.* The minimal hardware subsystems such as processing modules and primary memory modules are put together. In this phase, instruction-level debugging is carried out using the ROM-based monitor, perhaps supplemented by an in-circuit emulator. When the minimal hardware is operational, secondary memory and/or custom hardware may be added. Simple tests can be performed to establish that the hardware responds to commands. More complex tests for verifying the operation of the hardware must be written in assembler or in a system implementation language (e.g., PL/M-86 or C) and, in the absence of a working operating system, downloaded from a development station.

• *Operating system integration.* After configuration and generation of the operating system, it is booted or downloaded into the target memory. Getting an operating system operational in a not fully trusted hardware environment is best achieved using in-circuit emulation. Alternatively, the operating system may be started up in a number of phases using the monitor. On completion of each phase registers and data structures are checked.

• *Low-level system software/hardware integration.* The software for a real-time system typically contains several low-level, hardware-control functions which provide an abstract interface to higher software levels. These functions can be implemented as server tasks or as drivers integrated with the operating system. In either case, it should be possible to test and debug the hardware and associated control functions thoroughly so that they can be used by other system developers.

• *Module testing.* After designing and implementing a task, a programmer needs tools with which to test and debug it, preferably in isolation from other application tasks. Typically, an artificial environment is created for stimulating, controlling, and collecting responses from the task under test. In addition, stubs must be inserted for not-yet-existing tasks. In a multitasking environment, this involves creating ad hoc tasks which send messages to, and accept messages from the task under test. To be useful at this level, a debugger should provide facilities for interacting with the task on the message/mailbox level and for accessing all relevant data structures.

• *Application system integration.* The complete system is normally run for an extended period in the development environment before being released. During this period, the performance of the system is monitored, along with other important system parameters. It may be necessary to make small changes—such as fine-tuning task priorities—to satisfy response-time requirements.

• *Beta-site integration.* The system must be built up and interfaced to the customer's real world. Beta-site testing often reveals obscure bugs which may be nearly impossible to reproduce in a lab environment. The testing takes place in an environment of real customer needs. There will be strong pressure to keep the system available. To make things worse, access to the installed system may be severely restricted and only a subset of debug tools may be available. The dynamic debugger and/or static debug monitor can remain in the system (memory constraints permitting) during beta-site testing. During this phase of system testing, it is essential that the system development team be able to capture session histories and system status information for later analysis. It is also extremely useful if they can perform small ad hoc experiments and make minor adjustments to improve system performance.

Intel's debugging tools. Intel supplies an interesting combination of debugging tools, the most powerful being in-circuit emulation—I2ICE.[2] This tool provides symbolic debugging, intelligent breakpoints and tracing, and a versatile macro command language. I2ICE covers both the hardware level and the instruction level. It also allows access to the symbol tables generated by the language tools, thus

providing a degree of statement-oriented debugging.

However, I2ICE requires a host development station and emulation box permanently attached to the target hardware. This means that there typically will be only one target system running under in-circuit emulation, making the target system a scarce resource. In addition, the development station becomes unavailable to other users. For simple debugging, I2ICE is an expensive solution. Our experience is that in the early stages of development, I2ICE must be permanently connected to one system, which must be then dedicated to solving the most difficult debugging problems—usually those involving time-critical hardware. Once these problems have been surmounted, I2ICE is needed only for occasional troubleshooting.

The other Intel tools[1,3-5] are each targeted at a particular level and perform adequately. The main drawbacks of these tools are that

- they have no consistent common command language,
- they cannot be tailored easily to special programmer requirements,* and
- they provide no support for active programmer intervention or participation in the flow of control of the system.

In our experience, we found the last two drawbacks to be the most serious.

Debugging environment requirements

Interactive debugging. In our view, interactive debugging requires a complete environment as opposed to just a collection of tools. In such an environment, the programmer should have good visibility of, and wide-ranging control over, the physical and functional state of the system.[6,7] He should be able to communicate directly with all major objects—such as tasks—in the system. This programmer-to-object interaction should be the same as the object-to-object interaction; no special instrumentation in the object should be required.

The interactive debugging process models scientific method. The programmer first formulates one or more hypotheses about the system and then decides what must be observed and controlled to determine which, if any, hypothesis is valid.

Typically, tools to perform the observations and perturbations on the system do not already exist and must, therefore, be built. The process of creating these new tools should be a natural step in solving the problem at hand and should not require a major change of direction. The tools needed to solve particular problems may range from low-level, hardware-oriented ones to high-level, application-oriented ones.

It should be possible to create at least a primitive version of most tools by using the debugger's command language rather than a conventional programming language. However, whatever the language a particular tool is written in, be it the debugger's command language or, for example, C, it should be completely transparent. Control, debug, or status inspection functions which are already part of the system should be accessible as building blocks with which to construct new tools. We require a programmable debugging environment similar to the Unix shell programming environment.[8]

Interactive development. In complex, distributed processing applications, there is a limit to the progress that can be achieved without experimentation. In other words, interactive development is essential. A debugging environment can serve as a testbed for the construction and controlled execution of experiments in which the programmer can interactively create a new function and supply its parameters and variables. He can explore alternative designs to discover the implications of various combinations of design decisions and to gradually focus in on an appropriate design. The realm of interactive development can be expanded to include the user at a relatively early stage. For example, the debug testbed can be used to allow a proposed man-machine interface strategy to be tested at a low-risk phase in the project.

*SDM86—the system debug monitor—can be extended to provide facilities for static display and integrity checking of RMX86 data structures such as job and task tables, but no procedural access to the operating system is provided.

Monitoring and recording events or collecting data should be easy to implement in the debugging environment. This requires full access to all system resources and efficient means of communication with these resources.

Users. The users of a debugging environment are not only system programmers but also hardware engineers and applications specialists interested in collecting empirical data or modifying system parameters in order to optimize system operation. The debugger's command language should not only offer sufficient flexibility for programmers but should also be friendly toward other types of users.

In projects involving development of both software and hardware, a common debug and test language can serve as a valuable communication medium between developers in the various disciplines. Such a language can often replace or complement formal interface specifications.

On-target development. In on-target development, with its rapid edit-compile-link-debug cycle, fixing a bug or preparing command files using the standard operating system editor normally requires leaving the debugging environment and entering the programming environment. Returning to the debugging environment requires that it be fully reinitialized. For extensive programming activities this may be perfectly acceptable, but for browsing through source code or just looking at directories to locate a source file, it represents a major and unnecessary nuisance.

An integrated environment should provide a mechanism by means of which an activity can be temporarily suspended, another activity carried out, and the original activity resumed without loss of context. It should at least be possible to enter the programming environment and return to the debugging environment without significantly changing the debug state.

Interactive dynamic system building. Starting up a real-time system normally involves a system building step. During system building, the required jobs and tasks are loaded (if they are not already resident) and data structures are initialized. Then the tasks are activated and the system becomes operational. In the debugging environment, it should be possible to build the system interactively, starting with well-trusted components and adding less reliable modules one at a time. One should be able to delete a task, correct it, and load it again without significantly affecting the rest of the system. For this to be feasible, run-time binding between new tasks and the rest of the system must be employed. This can be implemented in various ways: for example, if a task communicates solely by message-mailbox or semaphore mechanisms it need only make its input mailbox publicly known at run time. This run-time binding can be implemented in a straightforward manner through object directories.[4] A task places a name and token for its mailbox in a preagreed object directory. Other tasks having access to this directory retrieve the token by supplying its name. This approach contrasts sharply with the static relations among components in single-tasking environments and, in fact, with the relations among components in many other multitasking operating systems.

Debugging distributed systems. Debugging a distributed processing system that uses multitasking in each processor is an art requiring considerable skill and experience from the programmer.[6,7,9-12] If possible, the same debugger should be available on each processor. It is essential to have a separate display area in which to present data relevant to each processor and not to mix data on one screen. In the absence of multiwindow interaction, this involves working at more than one keyboard.

On occasion, a particular debug control action must be started more or less simultaneously on a number of processors. Centralized debug control then becomes imperative. This requires that a debugger be able to assume temporarily the role of a "master" having a communication path to all the other "slave" debuggers so that it can issue coordinated system-wide commands. The slave debuggers should be dual-ported, i.e., be able to take their commands either from a terminal or from another source such as a server task responsible for communication with other processors. Consequently, each debugger should provide facilities for switching the input com-

mand stream to another stream supplied by a programmer-written task. The exact implementation is very system-dependent.

Dominant requirement. The dominant requirement for the debugging environment is that the debugger offer a simple but powerful command language with enough flexibility that extensions written in the command language or a conventional programming language can be easily implemented. These extensions should provide hooks to every level of the operating system or application software. Rather than offering general-purpose tools which cannot be modified, the debugging environment should provide primitives for building specialized tools. Creating a new tool from existing components should be perceived as a natural step rather than as an end in itself.

The debugging environment should be interactive not only in the sense that the programmer interacts with the debugger but also in the sense that he communicates at an appropriate level with the system-under-test in order to influence the flow of control.

Most debuggers in use today do not deal with these issues. The debugging environment which we have implemented meets many of the above requirements.

Fifth and its users

Rather than defining a new debug command language, we opted for the Forth approach.[13-15] Forth is a popular interpreted language which emphasizes interactive working, features extreme flexibility, and offers reasonable performance while requiring little overhead in the way of memory or system support.

The typical Forth environment is single-tasking, is implemented in assembler, and is used for developing applications. Our debugger runs mostly in multitasking systems, is implemented in a high-level language, and is used for interactive debugging. These differences are sufficiently radical that we felt that another name was justified. We called our debug language Fifth to acknowledge its debt to Forth.

We will take a look at how Fifth appears to its various users. (For information on the language itself, see "The kernel language," below.)

Main text continues on page 26

The kernel language

Basics. The basic language unit is called a "word." A word is any string of printable characters delimited by white space, i.e., spaces, tabs, or new lines. It is quite normal for words to be given graphic names such as a→d, wait&see, and ?error.

The "stack" is used for passing values between words. By convention, words remove their input arguments from the stack and return values on the stack. Adherence to this principle means that it is straightforward to test a word interactively at the keyboard. A set of words for manipulating the top few stack entries is provided.

The stack orientation means that commands are postfix—i.e., a command follows its arguments (if any). Careful choice of command names can lead to a very natural mode of interaction with the debugger:

```
100 test-loops
10 right-pulses pause 10 left-pulses
```

Numbers form a special class of word. Reading a word which represents a legal number in the current numeric base results in the corresponding value being pushed onto the stack. The numeric base can take any value between 2 (binary representation) and 40 (radix-40 representation).

Arithmetic operators treat stack entries as 16-bit signed integers, whereas logical and shift operators treat stack entries as 16-bit unsigned values.

Memory and I/O port access. A set of words provides direct access to memory locations and I/O ports. There are fetch (@) and store (!) words for 8-bit, 16-bit, and 32-bit fetch and store operations. Locations are represented as two 16-bit words on the stack. These form the base and offset of the

20-bit segmented address reference of the iAPX86 processor family.

Terminal input and output. Simple input and output words are provided to display numbers, print messages, fetch keystrokes, and obtain numeric values from the user.

Word definition. The "dictionary" contains a list of all the words currently known to Fifth. It can be extended with new words defined by the user, who thereby creates his own command set. The user can later remove ("forget") such definitions. The list of dictionary entries can be displayed.

A number of words create new definitions. Common to all such new definitions is that they themselves can be used in exactly the same way as "built-in" words.

Constants and variables. Constants are created by "const." When the newly created word is seen later, the constant value is pushed onto the stack. Variables are created by "var." When the new word is executed, the address of the associated storage cell is pushed onto the stack. Other words are provided which access the contents of such an address on the stack. Examples are

```
100  const portA              ( assign I/O address 100 to )
                              ( portA )
var  control-word             ( create the variable control- )
                              ( word )
ff   const reset              ( reset pattern )

reset control-word w!         ( reset control-word )

control-word w@ portA pw!  ( and copy to portA )
```

Colon definitions. The colon word signals the start of a compiled definition. Words appearing after a : token and the name of the new word are compiled into the new definition rather than immediately executed. A ; token terminates the compilation process. This procedure can be used as a sort of macro facility by which frequently used command strings can be abbreviated. An example is

```
: r1 10 right-pulses pause 10 left-pulses ;
```

This is the example in the "Basics" section, above, abbreviated to two letters.

In addition to the words available in direct execution mode, a number of control constructs may be used during the creation of a colon definition. These control structures can be nested; i.e., they can take the form of loops within loops, case statements within conditional branches, and so on.

Loops. The do...loop construct allows a loop to be repeated a specified number of times. The loop index start value and limit value are passed to "do" via the stack. Within the loop, the current loop index value is available as "i." The begin...end construct repeats the enclosed words until the argument passed to "end" becomes true (nonzero). Examples are

```
: 10-actions  10 0 do action pause loop ;
: idle  begin busy not end ;  ( wait till not busy )
```

Conditionals. The if...else...endif construct executes either the true branch (if...else) or the false branch (else...endif) depending on the truth value passed to "if." The case...endcase construct provides simple selection of a block enclosed by of...endof, or of a default block if no match is found. Examples are

```
: ?error status-word w@ if .text error! endif ;
                                    ( say so if error )

: ?status                           ( status word is on stack )
  case
       1  of .text out__of__range endof   ( error 1 )
       2  of .text phase__error   endof   ( error 2 )
      18  of .text bad__command endof
  endcase ;
```

Definers. Words can be defined which in turn can be used to define new words. This is useful for defining new classes or data types, where the differences between members of the class are determined at define-time, the execute-time behavior being identical. The pair of words "<builds" and "does>" is used to construct new defining words.

The entire construct is

```
: definer-name   <builds  ...define-time-actions...
                 does>    ...execute-time-actions...
```

The following example illustrates the building up of look-up tables (arrays of constants):

```
: table                    ( names a new defining word, table )
   <builds                 ( define-time actions: none )
   does>                   ( execute-time actions - )
     rot 2 * rot + swap    ( get location of nth integer entry )
     w@ ;                  ( get the constant stored there )

table power-of-2           ( create table )
1, 2, 4, 8,                ( store 4 constants into dictionary )

table power-of-10          ( another table )
1, 10, 100, 1000,

0 power-of-2               ( get first entry, i.e., 1 )
2 power-of-10              ( get third entry, i.e., 100 )
```

Data structure display and access. A common debug activity is inspecting operating system or application data structures. Rather than supplying words for accessing or displaying specific data structures, Fifth employs a general-purpose mechanism which allows the building of, display of, and access to any data structure anywhere in memory. The programmer supplies a descriptor template in the style of the format string taken by C's "printf" function.[1] A template reference is then supplied on each display or access:

```
text task-control-block
pc=%p\sp=%p\priority=%b\name=%s

( %b indicates a byte value in the current base )
( %s represents a string )
( %p represents a two-part pointer consisting of base and
      offset )
(\    is replaced by a space; no spaces are permitted inside
      a string )

my-tcb task-control-block printf
   ( display data structure )
pc=1234:567  sp=1234:abcd priority=128 name=my-task

: priority!   my-tcb " priority task-control-block store ;
                              ( change priority to TOS )

: priority@ my-tcb " priority task-control-block extract ;
                              ( push priority )
```

Commonly used templates can be stored in library files which can be read whenever access to a particular data structure is required.

Operating system interface. A number of facilities are available—or required—only if Fifth is running under an operating system.

Redirection of terminal input and output. Input redirection (only if Fifth is running under an operating system) is by means of the "get" word. Input can be redirected to a file or device—for example, to another serial port connected to a host or other debugger:

```
get /programmer/fred/myfile.5th
get :t1:
```

Under PMX86 it is possible to redirect to stream files. These files provide a means for two tasks to communicate with each other and do not use physical devices. An interesting application is to use an interprocessor communication server task to filter out debug commands and write them to a stream file. The Fifth input can then be redirected to this stream file, and it is then fully remotely controlled. Terminal output redirection is similar to terminal input redirection.

Session history. After lengthy debugging sessions, a developer often finds it useful to have a listing of all interactions for later analysis. If Fifth is running under an operating system, a session history file can be maintained. All terminal input and output is appended to this file. Session history can be switched on and off.

Reference

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

The hardware developer. A recurring problem in projects involving concurrent development of both custom hardware and software is that the hardware developers need at least minimal driver-level software at an early stage. However, at the same time the software team is working in a top-down fashion and is not yet ready or able to discuss "low-level details." The usual result is either that the (reasonable) requests of the hardware team are refused or that expensive ad hoc "test programs" are created by desperate hardware engineers.

Fifth provides low-level access to I/O ports and memory locations as well as easy mechanisms with which to build loops and conditional tests and to define symbolic constants and new definitions. These facilities allow the hardware engineer to exercise most of his subsystem, albeit at a fraction of real-time rates. Once the basic hardware operation has been established, the Fifth programs that have been developed can be used as the basis for extensions written in suitable systems programming language. These extensions allow full-speed operation. The Fifth programs can also provide valuable information supplementing formal hardware-to-software interface specifications.

The software developer. Classical approaches to module testing require the writing of stubs and the creation of test harnesses and data sets. Our approach is to allow simulation of the entire test environment—above (test harness), below (stubs), and test data—from within the Fifth environment. Employing the run-time binding philosophy described earlier allows tasks to be loaded from within a Fifth environment and exercised with data supplied from the keyboard, arbitrary test loops, and other tasks. Other tasks that would be present in a full system can be simulated.

Because of the ease with which even small, relatively "naked" tasks can be implanted in a test environment and loop-tested, a large number of bugs can be detected and corrected early in the development cycle.

The system builder. System building can be done interactively with Fifth. Each job or task can be loaded in turn and activated when required, and global objects (mailboxes, semaphores) can be interactively created, initialized, catalogued, and inspected. The system builder definitions can be collected in a file and executed by means of a terminal input redirection. The advantage of this is that the system building process can be tailored to a particular debug problem simply by editing this file—a much simpler process than is usually required for real-time system configuration or reconfiguration.

The prototype builder. Particularly in the area of the functional aspects of the man-machine interaction, it may be essential to see a proposed idea in action before committing the project to a specific choice.

Fifth supports the construction of prototypes in a fashion very similar to the support given the software developer. The aim in both cases is similar: to try out a potential component of a system in a partially simulated environment. In the case of prototyping, the critical software and hardware components can be prototyped and demonstrated without the massive investment which would be required to build a complete system.

The application specialist. During operation of the system, the application specialist will frequently require knowledge of various sytem parameters, particularly those determined by the system itself in response to its specific environment. He may also need to investigate the system's response to test stimuli, in order to investigate problems or shortcomings reported from the field.

Fifth, in peaceful coexistence with the application system, provides a trap door which allows the application specialist access to the system. And if a spare I/O channel is available, he may even have access while the system is otherwise in normal use. He may initiate calibration sequences, compare the results with expected values, and take appropriate action if the system is outside permitted tolerances.

Extensibility

We have emphasized the need for tool building to be natural and easy and not to

become an end in itself. The Fifth language itself offers many facilities for building tools interactively (see again "The kernel language"). However, in some cases access to application functions or the underlying operating system is required to build a new tool. It may be difficult to implement this access at the Fifth command language level, and so a different mechanism is needed.

Front end and back end. Fifth is implemented as a front end and back end. The front end consists of the Fifth interpreter and the standard built-ins. The back end is application-dependent and is written by the programmer in a system implementation language. The back end adds a number of programmer built-ins to the dictionary. These can be pre-existing functions for specific purposes, application functions, or routines which have to be written in C or PLM/86 for reasons of efficiency. A back end is also used to provide an interface to system calls of the underlying operating system. By providing access to the system calls, the programmer can influence the flow of control in the system. He can communicate with the system at the task level using the message-mailbox mechanism or by means of files. From his terminal he can, using standard system calls, create a message, look up a mailbox in an object directory, send the message to this mailbox, and then wait for the task's respone in the form of a response message or an action. Other features of the underlying operating system may also provide useful extensions such as access to the programming environment.

During initialization of the interpreter, a user-supplied start-up routine is called. This routine in turn can call an interpreter routine to enter one or more user functions as a new built-in.

User functions usually require a short interface routine to pop the arguments from the parameter stack, call the actual function, and push any results.

A sample back end is discussed in the box on page 28.

Adding new tools. The following summarizes the ways in which the programmer can add new debug tools to the basic vocabulary:

- He can type in a combination of existing commands ("words") with appropriate parameters and execute it immediately. If the function is required again, he must re-enter the entire command line.
- He can define a new word consisting of a combination of existing words. If execution of the function is required, he need only type in the name. On leaving Fifth, however, he loses the word definition.
- He can create a file containing new definitions which he can read in using the "get" word. Input stream redirection permits definitions to be from a file rather than from the keyboard. Files can be nested. Changes can be easily made by editing the files.
- He can create the new function in a system implemention language, compile it, and link it to Fifth as part of a back end. The new function becomes a Fifth primitive. Candidates for this treatment are functions that are so frequently required that the overhead involved in reading lengthy definition files becomes a nuisance, or functions that have to be implemented more efficiently than is possible in Fifth. Changes to the function can be made only by source recompilation and relinking.
- He can write the new function in a system implementation language, compile it, link it, and load it as a task or self-contained program (job). He can establish connections by means of the object directory and communicate by means of messages or semaphores. This mechanism allows considerable flexibility. He can easily edit the task from within Fifth (see again "RMX86 back end") and recompile and relink it. Task activation and deletion can be performed in a straightforward manner through standard system calls. Communication requires some special definitions that are independent of the task's current location in memory space.

Configurability

We have configured a range of customized Fifth systems for various applications. Fifth can be configured to run stand-alone or under an operating system. Under an operating system, Fifth runs as a job or task as determined by the programmer and the char-

RMX86 back end

The Fifth RMX86 back end provides an interface to most system calls of the nucleus, I/O system, loader, and human interface layers.[1] This allows the programmer to make system calls directly from his terminal and in this fashion interact with any currently active tasks.

Using normal Fifth mechanisms, a developer can easily define debug commands which send messages to specified mailboxes, suspend tasks, load jobs, create and access files, or send commands to the human interface.

System calls. An interface is provided to most system calls of the nucleus, I/O system, loader, and human interface layers:

```
0 create$mailbox const mbx
   ( create a mailbox and assign token )
      ( to constant mbx, mailbox flags=0 )
our-job mbx " mbx catalog$object
      ( and make it publicly known within )
      ( our job )

1 create$segment const msg
   ( create an uninitialized message object )
0 msg mbx send$message
   ( and send it to mbx mailbox, no )
      ( response is required )

0 10 0 create$semaphore const semaphore
      ( create semaphore with 0 initial units )
      ( and maximum of 10 units and flags=0 )

1 semaphore send$units
   ( send one unit to the semaphore )

100 1 semaphore receive$units .
      ( wait max. 100 system ticks for 1 unit )
```

Port to RMX86 human interface. RMX86 provides a "human interface" which normally interprets commands entered at the keyboard. Commands can also be issued from within a program. The human interface accepts commands via a command connection that is a memory-based data structure. By invoking the "send$command" call with a command connection and command string pointer as parameters, a developer can request that the command specified in the string be executed. He can use this feature to momentarily leave the debugging environment.

This facility ("run") can be used to browse through source files, list directories, and perform other actions normally reserved for the programming environment. Upon exit of the directory program, editor or whatever, control returns to Fifth. It is also possible to run another Fifth (perhaps with a different back end) from within Fifth:

```
" dir\/user/0/fred run
   ( sends the directory command string to )
      ( the human interface for execution. The )
      ( backslash "\" is replaced by a space as )
      ( no spaces are permitted inside a string )
```

This results in the running of the dir(ectory) command on /user/0/fred.

The "run" word is written as follows:

```
var cmd_con   ( command connection variable )

: run              ( expects cmd string on stack )
  c$create$command$connection cmd_con w!
     ( create connection )
  cmd_con w@ c$send$command
     ( send the command to connection )
  cmd_con w@ c$delete$cmd$con
     ( delete connection )
```

An alternative to this simple but somewhat clumsy method of invoking the human interface is to provide a back-end function which simply sends the rest of the line (possibly with continuation lines) to the human interface.

System building. System building can be performed interactively using Fifth and the application loader and nucleus calls[1] in the back end. Each job or task can be loaded in turn and activated when required:

```
" syscon.table   load-data
" my_task        install-task const my_task
                    ( returns a task token )
" my_job         install-job const my_job
                    ( loads and activates a job )
```

The words used in this example are defined in Fifth and use standard system calls.

Reference

1. Intel, *iRMX86 Programmer's Reference Manual*, Parts I and II, Intel Corp. Literature Distribution Dept., Santa Clara, CA.

acteristics of the operating system. It may be configured into the system or started later as a normal application program. Other configuration parameters are the size of the dictionary, the inclusion/omission of the session history, the specification of data structure access, and help features.

Let us look at some practical configurations in more detail. All these configurations were still in use at the time of this writing.

Stand-alone configuration. A small programmable system was required to assist in the development and testing of some special-purpose image processing hardware designed to form part of a larger system. The solution was Fifth in EPROM with a custom back end to communicate with the hardware. Definitions could be entered from the keyboard or via a parallel link from a remote development machine. An interface to the static debug monitor, which also resided in EPROM, was available, allowing easy transition between the two debug tools.

RMX88 versions. RMX88 is a small real-time operating system intended for use in embedded systems and dedicated controllers. Running Fifth as a task communicating with a spare serial I/O port allows other tasks in the system to be exercised, system structures to be inspected, priorities to be tuned, and so on, without halting the system. The back end provides access to executive calls and structures.

RMX88 versions have been used over the past few years as service trap doors in products and for a number of 8086-based controllers in prototype developments.

RMX86 versions. Running Fifth as a job—with a back end providing interactive access to (most) RMX86 system calls—allows interactive system building, task monitoring and exercising, and diagnosis to be performed at various levels.

RMX86 versions (with custom back ends) are being used in the development of complex multiprocessor real-time systems with special-purpose hardware.

The story at left describes the implementation of the RMX86 back end in detail.

Implementation

The project which gave rise to Fifth emphasized both rapid implementation and high reliability—normally conflicting aims. We reconciled these by designing into the architecture enough processing power that "optimization"—particularly coding in assembler—was not necessary.

The same philosophy was applied to the design of the interpreter. In practical terms this meant that it had to be written in a system implementation language and that the interpreter kernel had to be extended by routines written as procedures in the same language, with a simple interface to the kernel.

We considered porting one of the public-domain Forth implementions but eventually rejected this approach because of the large amount of assembler code that would have been involved and the number of modifications that would have been required to support high-level language extensions.

Our original Fifth version, comprising the kernel language and a back end for controlling the custom hardware, occupied a code space of about 12K. It was written entirely in PLM/86 and was developed in about two man-months, including writing of a (basic) user manual. Subsequent work has almost exclusively involved the creation and addition of custom back ends. Among these, the RMX88 back end (in PLM/86, involving an effort of about one man-week) and the RMX86 back end (in C, involving an effort of about one man-month) are noteworthy.

Our emphasis has tended to be on reliability rather than on speed. This bias is balanced by the ease with which specialized extensions can be added.

We have described a debugging environment which has been successfully used in a number of development projects by a variety of users. We have argued that debugging requires more than just a collection of tools and have outlined some requirements for a debugging environment.

The power of a debugger lies in its command language and its extensibility. Rather than offering specialized tools which cannot be

modified, a good debugger provides tool-building primitives. Fifth provides a very simple method for adding new primitives to the kernel system. These primitives can be created in the debug command language or in the programmer's favorite system implementation language. The interface with the system under test may range from being loosely coupled (message-mailbox interaction) to being very tightly coupled (procedural interface to operating-system- and application-level functions).

The most outstanding feature of Fifth is the access it gives to all system calls of the underlying operating system at terminal level. This provides the programmer with all the tools that he may need to communicate with objects in the system under test. It also provides, at minimal cost, interactive system building support and a simple port into the program environment through which the programmer can momentarily leave the debugging environment without significantly changing its state.

A frequently voiced criticism is that our use of reverse polish notation makes Fifth programs unreadable. However, this problem should not arise if tricky stack manipulation, lengthy definitions, and an excessive number of arguments are avoided. Short, well-factored definitions can be both readable and reusable.

Our experience has shown that most software bugs typical of multitasking and multiprocessing can be detected with our tools. Occasionally a static debug monitor or in-circuit emulator must be used when the system must be frozen or critical timing problems must be solved.

Customized versions of Fifth can be produced in a short time for specific applications, providing highly specialized and powerful tools in a standard framework.

The ideas we have presented here evolved during a highly successful project—the development of a digital imaging system based on several Intel processors. The basis of this success was the emphasis we placed on the availability of powerful tools during development. The first version of Fifth was available early in the project, in 1981, and included a back end for interactive control of a dedicated image processor. Fifth became very popular with the hardware engineers, most of whom would have been reluctant to use a compiled high-level-language for writing small test programs. They made extensive use of Fifth to send patterns or commands to hardware subsystems and to check the response on oscilloscopes or logic analyzers. Software developers used Fifth to debug tasks, to test combinations of tasks, or to quickly create diagnostic tests. Application specialists used Fifth to calibrate the X-ray system and TV chain. Our original intention was to remove Fifth from the production version of the imaging system. Strong pressure from factory and service personnel convinced us that it should permanently remain in the system.

Other development groups in our organization have received unsupported source copies of Fifth. All these groups have successfully integrated it into their own development environments and have tailored it to their specific requirements. The present Fifth user community exceeds 40 people; at least 10 are fluent in using its many features.

Fifth provides an environment in which testing and debugging time can be significantly reduced. That benefit alone has justified the time we invested in its development.

Acknowledgments

We thank the hardware and software staff of the DVI project at Philips Medical Systems for their many useful ideas and suggestions. We also thank Julie Wilson, whose critical comments significantly improved this article.

Requests for copy of sources

Requests for Fifth sources (on RMX86 floppy disk or VAX tape) should be directed to Ms. Lieve Brauns, Philips Medical Systems, Dept. PG1, Building QJ1, Veenpluis, Best, Holland.

References

1. Intel, *iPSCOPE,* Intel Corp. Literature Distribution Dept., Santa Clara, CA.
2. Intel, *I2ICE—Integrated In-Circuit Emulation,* Intel Corp. Literature Distribution Dept., Santa Clara, CA.

3. Intel, *iRMX86 Introduction and Operator's Manual,* Intel Corp. Literature Distribution Dept., Santa Clara, CA.

4. Intel, *iRMX86 Programmer's Reference Manual,* Parts I and II, Intel Corp. Literature Distribution Dept., Santa Clara, CA.

5. Intel, *iSDM86 System Debug Monitor Reference Manual,* Intel Corp. Literature Distribution Dept., Santa Clara, CA.

6. H. K. Berg and M. G. Smith, "A Distributed System Experimentation Facility," *Proc. 3rd Int'l Conf. on Distributed Computing Systems,* Miami, Florida, 1982, pp. 324-329.

7. D. Bhatt and M. Schroeder, "A Comprehensive Approach to Instrumentation for Experimentation in a Distributed Computing Environment," *Proc. 3rd Int'l Conf. on Distributed Computing Systems,* Miami, Florida, 1982, pp. 330-339.

8. B. W. Kernighan and R. Pike, *The Unix Programming Environment,* Prentice-Hall, Englewood Cliffs, NJ, 1984.

9. H. Garcia-Molina et al., "Debugging a Distributed Computing System," *IEEE Trans. Software Engineering,* Vol. SE-10, No. 2, Mar. 1984, pp. 210-219.

10. R. Curtis and L. Witti, "BugNet: A Distributed Applications Debugging System," *Proc. ACM Sigsoft/Sigplan Symp. on High-Level Debugging,* Mar. 1983, Pacific Grove, CA.

11. W. C. Gramlich, "Checkpoint Debugging," *Proc. ACM Sigsoft/Sigplan Symp. on High-Level Debugging,* Mar. 1983, Pacific Grove, CA.

12. K. Marzullo, "A Note on Debugging Distributed Systems," *Proc. ACM Sigsoft/Sigplan Symp. on High-Level Debugging,* Mar. 1983, Pacific Grove, CA.

13. R. G. Loeliger, *Threaded Interpretive Languages,* Byte Books, Peterborough, NH, 1981.

14. L. Brodie, *Starting Forth,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

15. P. M. Kogge, "An Architectural Trail to Threaded-Code Systems," *Computer,* Vol. 15, No. 3, Mar. 1982, pp. 22-32.

For further reading

F. W. van der Linden and I. M. Wilson, *Fifth User Guide,* internal document, Philips Medical Systems Div., Best, Holland, 1982. (Available on the release medium. See "Requests for copy of sources," above.)

F. W. van der Linden and I. M. Wilson, "Real-Time Executives for Microprocessors," *Microprocessors and Microsystems,* Vol. 4, No. 6, Aug. 1982, pp. 211-218.

Frits van der Linden is manager of real-time operating system development at Zilog in Campbell, California. Previously, he was a project leader at Philips Medical Systems, Holland. His research interests include software engineering and development and distributed real-time operating systems. Van der Linden received his MS from the Technical University of Delft, Holland, and his PhD in electrical engineering from the University of Essex, England. He is a member of the IEEE Computer Society and of the ACM.

Van der Linden's address is Zilog, 4129 Park Blvd., Palo Alto, CA 94306.

Ian Wilson is a staff engineer with Philips Ultrasound. Prior to this, he was a consultant at BSO/Automation Technology, a Dutch company specializing in software for real-time and high-technology systems. For the past few years, he has been involved primarily with the design and development of medical imaging systems. His technical interests include design of real-time systems, interactive software development, the hardware/software interface, and medical applications.

Wilson began his career in 1973 when he joined the GEC Hirst Research Centre in London. He transferred to GEC Medical Ltd. in 1976. In 1978 he joined Scicon Consultancy International Ltd. Wilson received his MA from Jesus College, Oxford, England, in 1973.

Wilson's address is Philips Ultrasound, 2722 South Fairview St., Santa Ana, CA 92704.

A Performance Study of Mutual Exclusion/Synchronization Mechanisms in an IEEE 796 Bus Multiprocessor

E. Pearse O'Grady
Arizona State University

Raul Lozano
Data/Ware Development, Inc.

Of three mechanisms evaluated, the FIFO semaphore tied up the system bus the least.

The microprocessor provides the means for developing special-purpose multiprocessor and parallel processor systems which can focus the power of many processors on the solution of a single problem. Incorporating multiple processors into a single system introduces the problems of synchronizing cooperating processors and providing mutual exclusion among processors competing for common resources. These problems have been studied in the context of single-processor systems[1] and more recently in the context of multiple microcomputer systems.[2] They can be handled by hardware and software mechanisms such as flags, semaphores, and monitors. However, there is an overhead associated with implementing these mechanisms. Here, we describe a study conducted to quantify and characterize this overhead in a system in which multiple processors interact via a shared IEEE 796 system bus (Multibus).

The study supported the design of a parallel processor system for continuous system simulation applications[3]; this system requires synchronization and mutual exclusion mechanisms at different levels in its hierarchy. Efficient hardware mechanisms can be readily incorporated into the designs of the special-purpose processors being developed for the system, but system components implemented with off-the-shelf single-board computers are constrained by the design of the boards and generally require mechanisms implemented in software or in a combination of software and available hardware.

We evaluate three mechanisms in terms of a reporting problem in which processors simultaneously request mutually exclusive access to a shared memory. The results graphically demonstrate the effects of bus contention in a Multibus-based multiprocessor and the relative merits of flag and semaphore

0272-1732/85/0800-0032$01.00 © 1985 IEEE

mechanisms for providing mutually exclusive access to shared resources.

System configuration

Figure 1 illustrates the system used in the study; it is a multiprocessor consisting of four Intel iSBC 86/12A single-board computers interconnected by a single Multibus. One computer (SBC0) functions as the system control processor; the others (SBC1, SBC2, and SBC3) function as processing elements (PEs). Parallel priority resolution logic determines which requesting computer has the highest priority for gaining control of the bus when simultaneous requests occur. Priorities are arranged in sequence so that SBC3 has the highest priority and SBC0 the lowest. (See the overview of the iSBC 86/12A and the Multibus, on page 34.)

The reporting problem

In general, the study consisted of tests which have the following form. Through a dialogue with the user, SBC0 initializes the system to carry out one of four tests. SBC0 then interrupts the three PEs, causing each to execute its test program. The test programs call for each PE to enter a loop in which it decrements a counter from an initial value to zero. When the counter reaches zero, the PE reports the completion of its task to SBC0 by incrementing a count value (labeled DONE) in the dual-port memory of SBC0 and placing the resulting count value in a location in SBC0 that is associated with the PE (labeled LOC*x*, where $x = 1$, 2, or 3 corresponding to the PE). The program segment for a PE's reporting process is

```
INC   ES:BYTE PTR [DONE]
MOV   AL,ES:BYTE PTR [DONE]
MOV   ES:BYTE PTR [LOCx],AL
```

This requires each PE to perform four Multibus cycles:

(1) read DONE from SBC0 to PE*x*,
(2) write the incremented count value from PE*x* to DONE,
(3) read DONE from SBC0 to PE*x*, and
(4) write the count value from PE*x* to LOC*x*.

See "The reporting process," page 40, for more details.

If SBC0 initializes the value in DONE to zero, a value of 3 in DONE indicates that all three PEs have reported completion of their tasks; alternately, if LOC1, LOC2, and LOC3 are initialized to zero, nonzero values in all three locations indicate completion of all tasks. After completion of all tasks, each of the values 1, 2, and 3 is in one and only one of the locations LOC1, LOC2, and LOC3, indicating the order in which the PEs reported

cont'd on page 38

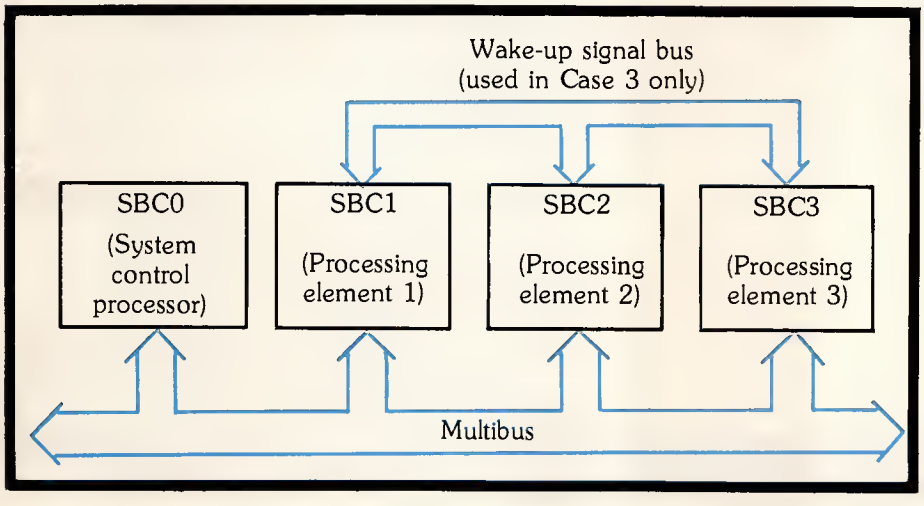


Figure 1. System configuration.

The iSBC 86/12A and the Multibus

E. Pearse O'Grady

The iSBC 86/12A,[1,2] illustrated in Figure 1, includes a 5-MHz 8086 microprocessor, 32K bytes of read/write memory (RAM), a priority interrupt controller, a programmable timer, programmable serial and parallel I/O ports, a dual-port memory controller, a Multibus interface, and up to 16K bytes of EPROM. A local bus interconnects all on-board memory and I/O ports for local memory and I/O operations; it accesses the system bus (i.e., the Multibus) through the Multibus interface for all nonlocal memory and I/O operations.

The dual-port control logic receives requests for access to the RAM from the CPU (i.e., the 8086) and from the Multibus, and grants access to it on a cycle-by-cycle basis according to the following strategy. If the RAM is free and a single request is received, the request is granted. If the RAM is busy, granting of the request is held off until the RAM is released. If simultaneous requests are received, the CPU is granted control of the RAM, and the granting of the Multibus request is held off until the CPU releases the RAM. When its dual-port RAM is accessed from the Multibus, the iSBC 86/12A appears to the Multibus to be a slave memory board.

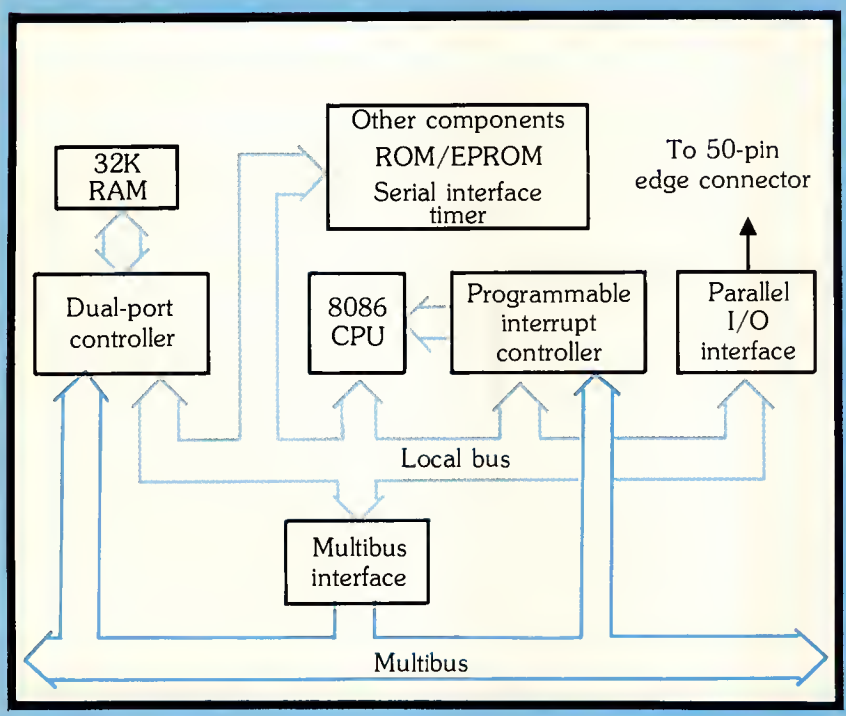


Figure 1. iSBC 86/12A single-board computer.

The Multibus[3] is a bus structure developed by Intel for interconnecting printed-circuit boards in microcomputer-based systems. The bus continues to evolve through extensions to the original structure; Johnson and Kassel[4] discuss the current version, Multibus II. The IEEE Std 796 bus is a version of the Multibus specified in a standard approved by the IEEE Standards Board in December 1982.[5,6] The Multibus interface of the iSBC 86/12A complies with IEEE Std 796 bus specifications with one exception, discussed below.

The IEEE Std 796 bus supports memory and I/O data transfers, direct memory access, interrupt generation, and other operations needed in a microcomputer-based system. In addition to power lines and reserved signal lines, there are 66 bus signals grouped in the following classes:

- control lines (8),
- address (24) and address-related (3) lines,
- data lines (16),
- interrupt (8) and interrupt-acknowledge (1) lines, and
- bus exchange lines (6).

In general, printed-circuit boards connected to the bus are either masters or slaves. Masters (e.g., single-board computers such as the iSBC 86/12A) in a multimaster system contend for control of the bus by issuing specified bus exchange signals. Various priority schemes are available for determining when a master is to surrender control of the bus and which master is to take control of the bus when control is exchanged between

masters. Slaves (e.g., memory expansion boards) respond to commands received from masters and have no capability to control the bus.

The IEEE Std 796 bus supports asynchronous memory and I/O read and write operations. Figure 2 illustrates the timing relationships among bus signals during a memory-write operation. After gaining control of the bus, the master places address and data onto the address lines (A0*-A23*)† and data lines (D0*-D15*) and asserts the memory-write command signal (MWTC*). The addressed slave memory stores the data and asserts the transfer-acknowledge signal (XACK*) to inform the master that the write operation has been completed. On receiving the XACK* signal, the master removes the address, data, and command from the bus and continues execution. Similar handshaking operations with the command signals MRDC*, IORC*, and IOWC* perform memory-read operations and I/O read and write operations.

Figure 3 illustrates a parallel priority bus arbitration scheme. To gain control of the bus, a master asserts its bus request signal (BREQ*). The parallel priority resolution logic receives request signals from all masters, determines which requesting master has the highest priority, asserts a bus-priority-in signal (BPRN*) to that master designating it as the highest-priority requester, and negates the bus-priority-in signals to all other masters. The designated master monitors the

†An asterisk (*) following a signal name indicates an active-low signal.[6]

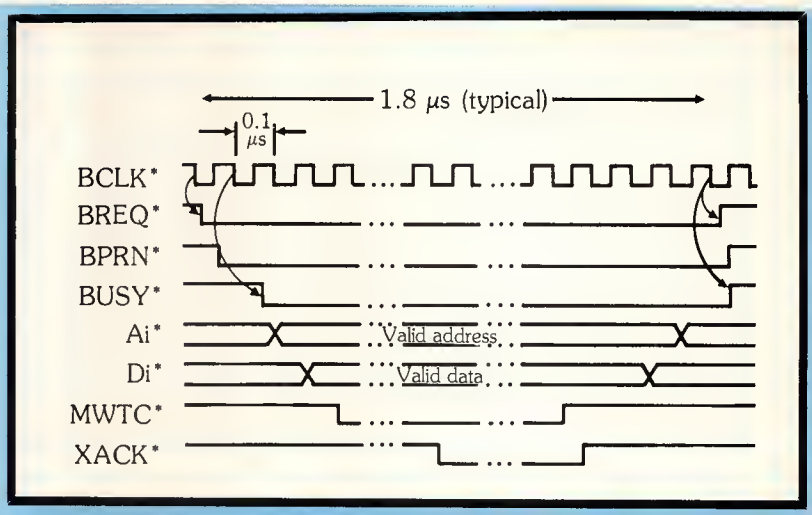


Figure 2. Timing for IEEE Std 796 memory-write operation.

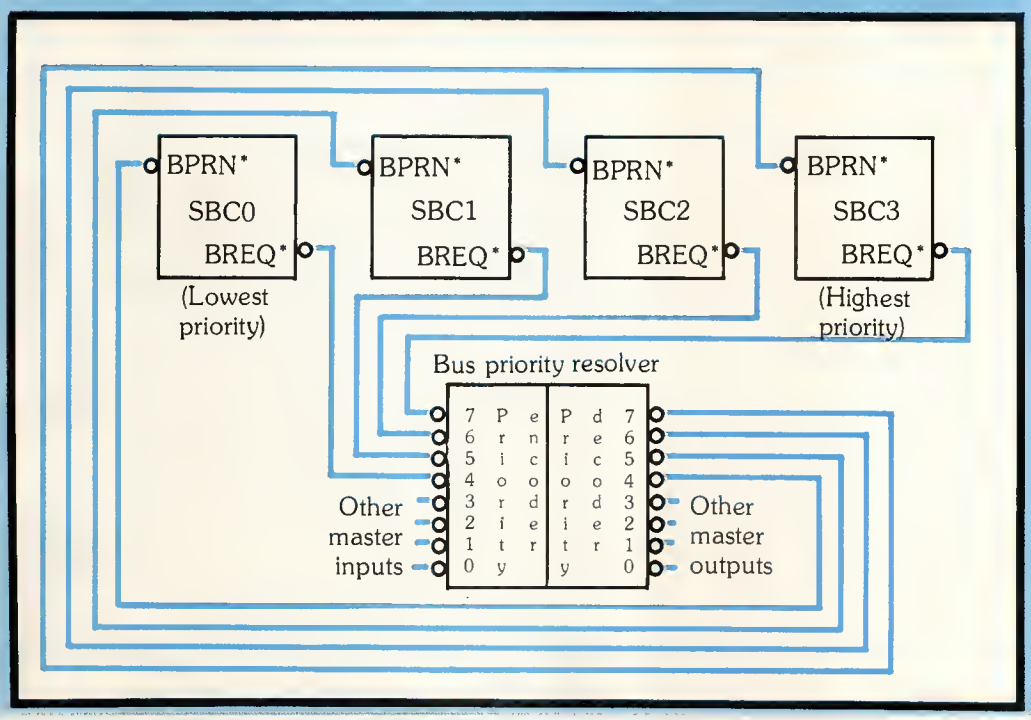


Figure 3. Parallel priority resolution logic.

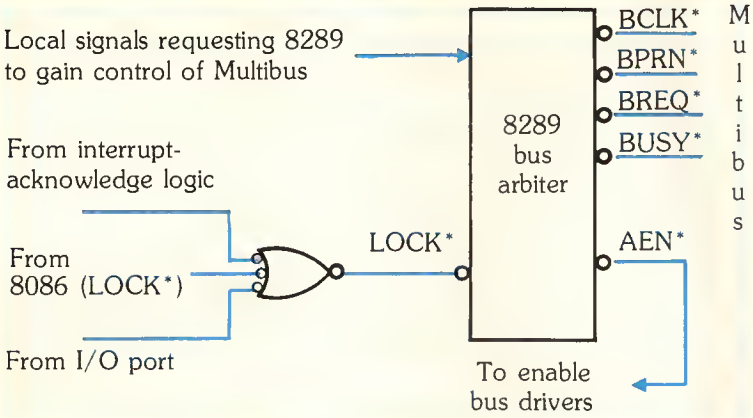


Figure 4. 8289 bus arbiter signals.

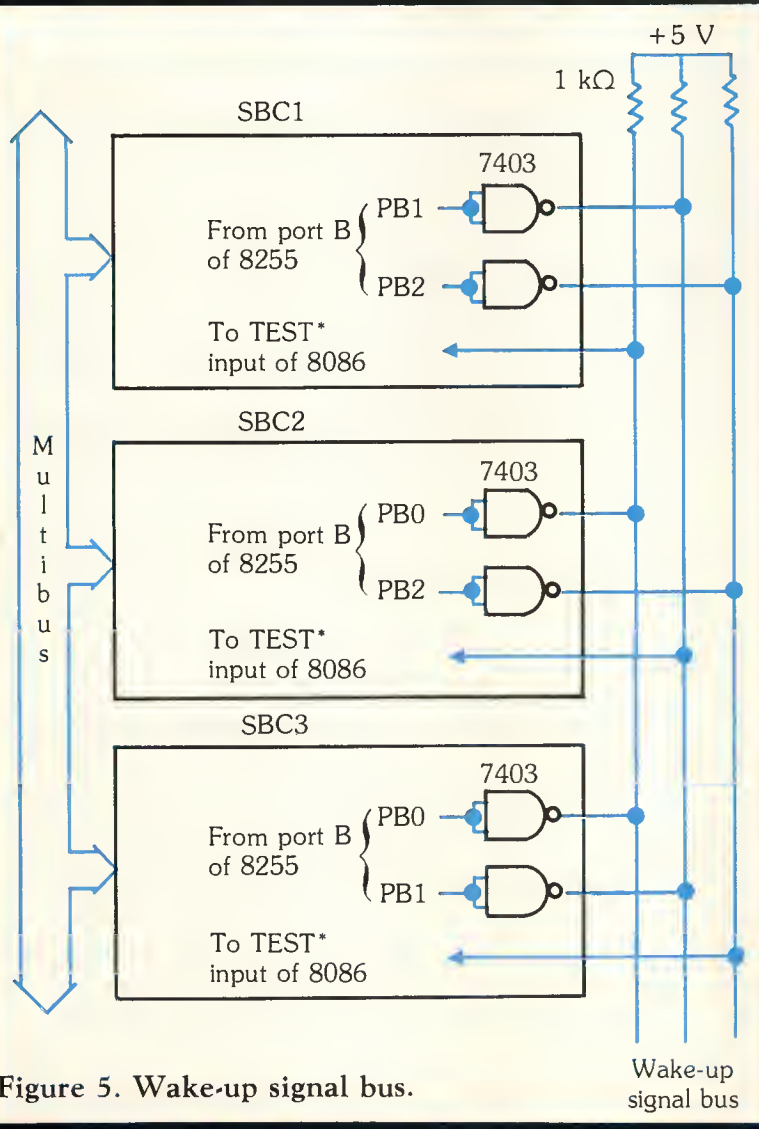


Figure 5. Wake-up signal bus.

bus-busy signal (BUSY*) to determine if the bus is free or if some other master is using it. If BUSY* is asserted, indicating that the bus is busy, the designated master waits until the current master releases the bus. When the bus is free, the designated master takes control of the bus by asserting BUSY* and retains control until it releases the bus by negating BUSY*. The BREQ* signals and the BUSY* signal are synchronized by the high-to-low transitions of a 10-MHz bus clock (BCLK*). Priorities are resolved during each bus clock period; when the bus is busy, the identity of the designated master can change during successive bus clock cycles if a sequence of requests, each of a higher priority than the last, is received.

A specialized device, the 8289 bus arbiter[7] (Figure 4), processes bus exchange signals on the iSBC 86/12A. In our study the bus arbiters are configured to surrender control of the Multibus after each bus cycle unless the processor is performing an operation with the bus locked. When a processor locks the bus, it retains continuous control of the bus for a number of Multibus cycles by asserting the BUSY* signal continuously from the start of its first bus cycle to the end of its last bus cycle. A bus cycle is an interval during which a memory or I/O read or write operation or an interrupt-acknowledge operation occurs. Locked bus operation is referred to as bus override in IEEE Std 796. The iSBC 86/12A provides two methods for performing locked bus operations. If an assembly language instruction requiring access to the Multibus includes a one-byte LOCK prefix,[8] the bus is locked during execution of the instruction. If it is necessary to lock the bus during execution of a number of instructions, jumpers can be installed so that, if the program clears a bit in the parallel I/O port, the bus arbiter will retain control of the Multibus from the start of its next Multibus cycle until the bit in the I/O port is set. With either method the LOCK* input to the 8289 is asserted during the interval in which the bus is locked.

An 8259A programmable interrupt controller[7] handles eight vectored priority interrupt requests on the iSBC 86/12A. Through jumper connections on the board,

the user selects the sources of interrupt requests from a variety of possible sources including the eight prioritized interrupt request lines of the Multibus (INT0*-INT7*). In responding to interrupts, the 8086 issues two interrupt-acknowledge pulses. The first freezes the state of interrupt requests for priority resolution and the second transfers a one-byte vector address to the 8086. Depending on the initialization of the 8259A, the address is supplied by the 8259A or from a different source such as the Multibus. Because the source of an interrupt generally is not known in advance, the iSBC 86/12A always acquires control of the Multibus and locks it while the two interrupt-acknowledge pulses are active. These pulses drive the INTA* signal line of the Multibus. In our study, the 8259As of the processing elements are configured to respond to interrupt requests received on the INT5* line and are initialized to supply a vector address to the 8086.

An 8255A programmable peripheral interface (PPI)[7] provides 24 parallel input or output lines on the iSBC 86/12A. These lines can be connected through buffers to a 50-pin connector. This connector is not covered by IEEE Std 796 or Multibus specifications. In Case 3 of the study, three I/O lines are bussed between processing elements to form a wake-up signal bus. On each processing element two PPI outputs connect through open-collector drivers to the connector and drive this bus as illustrated in Figure 5; the third bussed line, which is driven by the other processing elements, is jumpered to the TEST* input of the 8086. The voltage levels on the bussed signal lines are normally high. A processor issues a wake-up pulse on one (or both) of its output lines by writing a one to the corresponding PPI bit, driving the bussed signal low, and then writing a zero to the PPI bit, bringing the bussed signal high again.

As noted above, the Multibus interface of the iSBC 86/12A complies with the IEEE Std 796 bus specification with one exception. Compliance is at the highest level (viz., 8/16-bit data path, 16-bit I/O address path, and both bus-vectored and non-bus-vectored interrupts with level or edge-level triggering) in all respects except address path width. The standard provides 16-bit, 20-bit, and 24-bit levels of compliance for address path width; the iSBC 86/12A operates with a 20-bit address. The exception is that the iSBC 86/12A does not support the function associated with the LOCK* line of the IEEE 796 bus. This line enables the bus master to lock the dual-port memory of a bus slave when the master requires exclusive access to it during a sequence of memory cycles. The BUSY* line allows for this mutual exclusion in controlling the bus; the LOCK* line allows mutual exclusion to be extended off of the bus. The dual-port memory of the iSBC 86/12A cannot be locked from the Multibus. For this reason, in the tests, variables to which the processing elements require mutually exclusive access are located in the dual-port memory of the system control processor rather than in that of one of the processing elements.

References

1. R. Garrow et al., "16-bit Single-Board Computer Maintains 8-bit Family Ties," *Electronics*, Oct. 12, 1978.
2. *iSBC 86/12A Single Board Computer Hardware Reference Manual*, Intel Corp., Santa Clara, CA, 1981.
3. *Intel Multibus Specification*, Intel Corp., Santa Clara, CA, 1981.
4. J. B. Johnson and S. Kassel, *The Multibus Design Guidebook: Structures, Architectures, and Applications*, McGraw-Hill, New York, 1984.
5. R. W. Boberg, "Proposed Microcomputer System 796 Bus Standard," *Computer*, Vol. 13, No. 10, Oct. 1980, pp. 89-105.
6. *IEEE Std 796-1983: IEEE Standard Microcomputer System Bus*, IEEE, New York, 1983.
7. Y. Liu and G. Gibson, *Microcomputer Systems: The 8086/8088 Family*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
8. R. Rector and G. Alexy, *The 8086 Book*, Osborne/McGraw-Hill, Berkeley, CA, 1980.

to SBC0. In order for the reporting mechanism to operate correctly, only one PE at a time can execute the sequence of steps 1 to 3; thus, there is a requirement for mutual exclusion in accessing location DONE.

Simulating the task performed by a PE with a counting loop allows tasks of different durations to be assigned to the PEs. However, the most interesting situation—the one considered in the study—occurs when all PEs execute tasks of the same duration and attempt to report simultaneously.

Synchronization and mutual exclusion mechanisms

There are three techniques for providing mutual exclusion during the reporting process. In one, the reporting PE locks the bus during its reporting process. In the second, a software flag controls the sequence in which PEs report. In the third, a semaphore implemented by a combination of hardware and software controls the sequence. Below, we examine

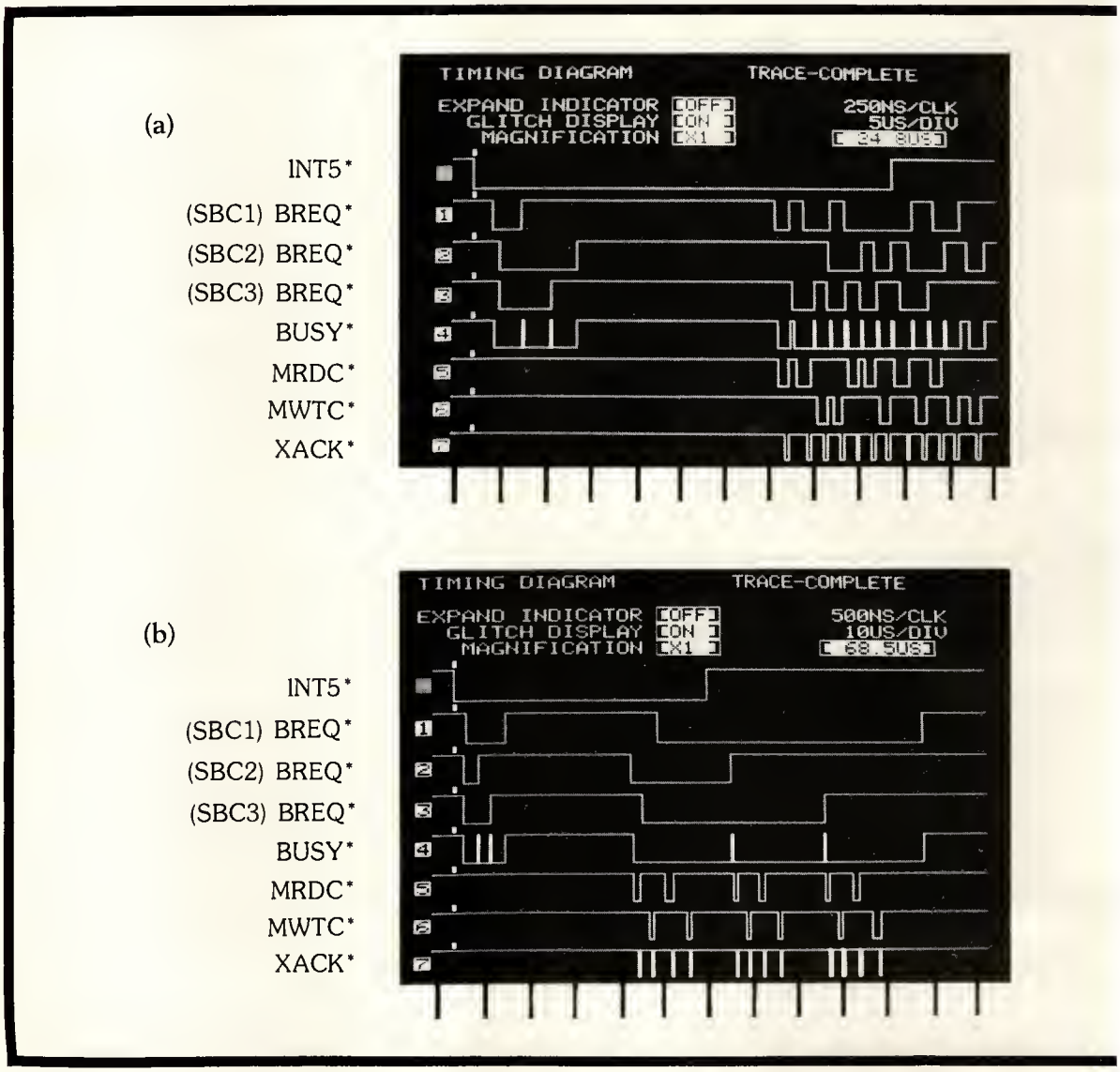


Figure 2. Timing diagrams for four cases: (a) Case 0—no mutual exclusion, (b) Case 1—locked bus, (c) Case 2—flag, and (d) Case 3—FIFO semaphore. Time divisions are marked below each photo; time-per-division information is included in the photo. The

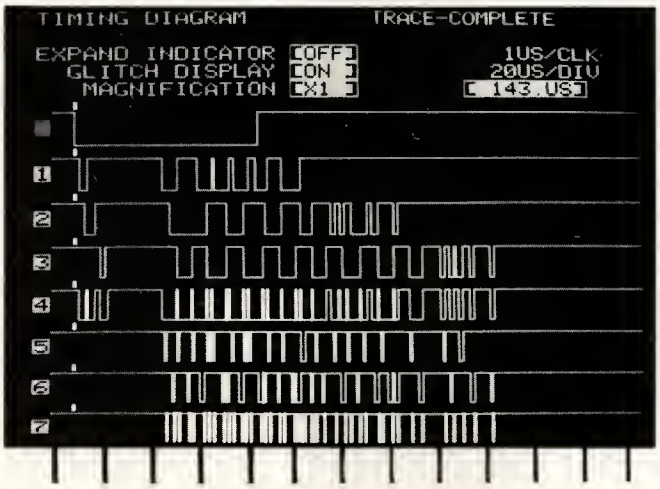
these techniques, as well as the effects of not providing mutual exclusion.

Case 0: No mutual exclusion. In this case there is no mechanism for providing mutual exclusion during the reporting process. When a PE completes its task, it begins reporting to SBC0 without regard for whether other PEs are reporting at the same time. If the PEs begin reporting at about the same time, their Multibus cycles are interleaved in an unpredictable sequence during the reporting process. In general, the reporting process is not performed correctly in this case. For example, it is possible for all three PEs to read DONE before any PE writes the incremented count value back to DONE, resulting in DONE having a final value of 1 rather than 3. The final values of the LOC*x* variables will also be incorrect in this case.

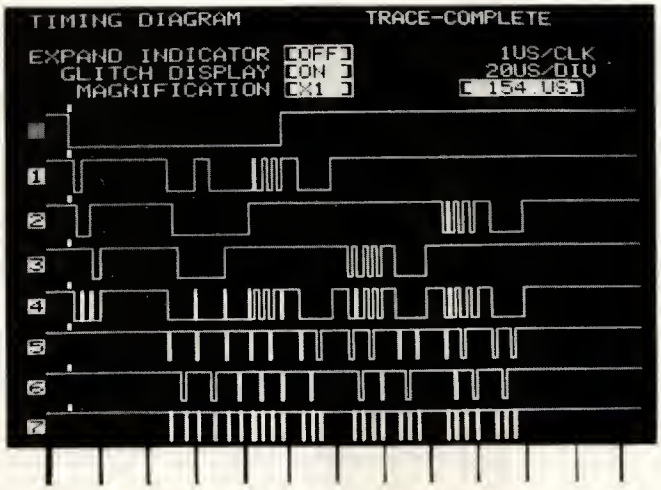
The timing diagram of Figure 2a describes Multibus activity during a typical run. The left

Cont'd on page 43

(c)



(d)



time shown in reverse video is the duration of the reporting period (from the start of the first bus request to the end of the last bus request) for all three PEs. The traces are recorded on a Hewlett-Packard 1615A logic analyzer.

The reporting process

Figure 1 lists the program segment implementing the reporting process for SBC2. Implementation of the cases using no mutual exclusion, a locked bus, and the flag mechanisms is straightforward. In the FIFO semaphore implementation, the four high-order bits of FIFO (a location in the dual-port memory of SBC0) hold the count, encoded as the complement of the number of PEs waiting on the resource. An all-zero bit pattern in FIFO indicates the resource is free and there are no pending requests. The twelve low-order bits of FIFO form a bit map which identifies the PE controlling the resource and the PEs waiting on the resource. For the run shown in Figure 2d in the main text, the following sequence of hexadecimal values is stored in FIFO during the run:

```
;       ON ENTRY:  AH = 1; BL = 0, 1, 2, OR 3 AND IDENTIFIES THE LOCKING
;       MECHANISM TO BE USED; ID2 EQU 0F002H.
;
;       ENTRY FOR FLAG MECHANISM (CASE 2)
FLG1  LOCK XCHG  AH,ES:BYTE PTR [FLAG]   ; TEST AND SET FLAG.
      TEST       AH,AH                   ; USES LOCK PREFIX
      JNZ        FLG1                    : BUSY, TRY AGAIN
      JMP        RPT1                    ; NOT BUSY, REPORT
;
;       ENTRY FOR LOCKED BUS MECHANISM (CASE 1)
LCK1  MOV        AL,08H                  : SET I/O BIT TO LOCK BUS
      OUT        0CEH,AL
;
;       ENTRY FOR NO MUTUAL EXCLUSION MECHANISM (CASE 0)
RPT1  INC        ES:BYTE PTR [DONE]      ; REPORT TO SBC0
      MOV        AL,ES:BYTE PTR [DONE]
      MOV        ES:BYTE PTR [LOC2],AL
      CMP        BL,2
      JNE        FIN1
      MOV        ES:BYTE PTR [FLAG],0    ; RESET FLAG IN CASE 2
;
FIN1  MOV        AL,09H                  ; RESET I/O TO UNLOCK BUS
      OUT        0CEH,AL
      JMP        EXIT
;
;       ENTRY FOR FIFO SEMAPHORE MECHANISM (CASE 3)
```

Figure 1. Program segment for SBC2's reporting process. Identical program segments are used in SBC1 and SBC3 except that the expression name ID2 is replaced by ID1 or ID3, which have values 0F001H and 0F004H, respectively, and the variable LOC2 is replaced by LOC1 or LOC3. Listings are in the 8086 assembly language of the Hewlett-Packard 64000 development system.

0000 (initial value),
F001 (after SBC1's P operation),
E005 (after SBC3's P operation),
D007 (after SBC2's P operation),
E006 (after SBC1's V operation),
F002 (after SBC3's V operation), and
0000 (after SBC2's V operation).

Figure 2 is a timing analyzer trace showing the Multibus signals generated as SBC1 carries out its reporting process under the conditions that no mutual exclusion mechanism is used (i.e., Case 0) and SBC1 is the only processing element reporting. Four bus cycles occur. The first two are generated by the instruction which increments DONE: INC ES:BYTE PTR [DONE]. In the first bus cycle, MRDC* is asserted as SBC1 reads the initial value of DONE from the dual-port memory of SBC0. Between the first and sec-

```
FIF1  MOV     BX,#ID2                  ; BX = 0F002H
      MOV     AX,0908H                 ; SET I/O BIT TO LOCK BUS
      OUT     0CEH,AL
      ADD     BX,ES:WORD PTR [FIFO]    ; DECREMENT COUNT AND
      MOV     ES:WORD PTR [FIFO],BX    ; INSERT ID IN BIT MAP
      MOV     AL,AH                    ; RESET I/O TO UNLOCK BUS
      OUT     0CEH,AL
      JNC     ENL1                     ; IF RESOURCE FREE, REPORT
SL1   SUB     BH,0F0H                  ; IF RESOURCE BUSY, COMPUTE
      CMP     BH,0F0H                  ; POSITION ON WAITING LIST
      JE      EL1                      ; AND WAIT FOR WAKE-UP PULSE
      WAIT                             ; WAIT HERE IF NOT AT TOP
      JMP     SL1
EL1   WAIT                             ; WAIT HERE IF AT TOP OF LIST
;
ENL1  INC     ES:BYTE PTR [DONE]       ; REPORT TO SBC0
      MOV     AL,ES:BYTE PTR [DONE]
      MOV     ES:BYTE PTR [LOC2],AL
;
      MOV     AX,0908H                 ; SET I/O BIT TO LOCK BUS
      OUT     0CEH,AL
      SUB     ES:WORD PTR [FIFO],#ID2  ; INCREMENT COUNT AND
      MOV     BX,ES:WORD PTR [FIFO]    ; REMOVE ID FROM BIT MAP
      MOV     AL,AH                    ; RESET I/O TO UNLOCK BUS
      OUT     0CEH,AL
      JNC     EXIT                     ; EXIT IF NO PEs WAITING
      MOV     AL,BL                    ; ISSUE WAKE-UP PULSES IF
      OUT     0CAH,AL                  ; ANY PEs ON WAITING LIST
      XOR     AL,AL
      OUT     0CAH,AL
      JMP     EXIT
```

ond bus cycles, the value is incremented within the CPU of SBC1. In the second bus cycle, MWTC* is asserted as SBC1 returns the incremented value to DONE. The instruction MOV AL,ES:BYTE PTR [DONE] generates the third bus cycle; MRDC* is asserted as the new value of DONE is read from the dual-port memory of SBC0 to the AL register within the CPU of SBC1. The instruction MOV ES:BYTE PTR [LOC1],AL generates the final bus cycle; MWTC* is asserted as SBC1 writes the value from its AL register into LOC1 in the dual-port memory of SBC0.

SBC1 requests control of the bus prior to each bus cycle by asserting its bus request signal, (SBC1) BREQ*. Because no other requests are being made, SBC1 has the highest-priority active request and takes control of the bus after one bus clock cycle (i.e., after 100 nanoseconds) by asserting BUSY*. The dual-port memory of SBC0, functioning as a slave memory, responds to memory read and write commands issued by SBC1 by asserting XACK*.

The different durations of the bus cycles in Figure 2 reflect the effects of contention for access to the dual-port memory of SBC0. This contention arises from concurrent requests by a Multibus master (viz., SBC1) and by the CPU of SBC0. The first bus cycle is clearly longer than the others. This is due to SBC1's request for access to the dual-port memory being held off while the CPU of SBC0 completes a memory cycle. In the absence of this contention, the duration of each bus cycle would be the same.

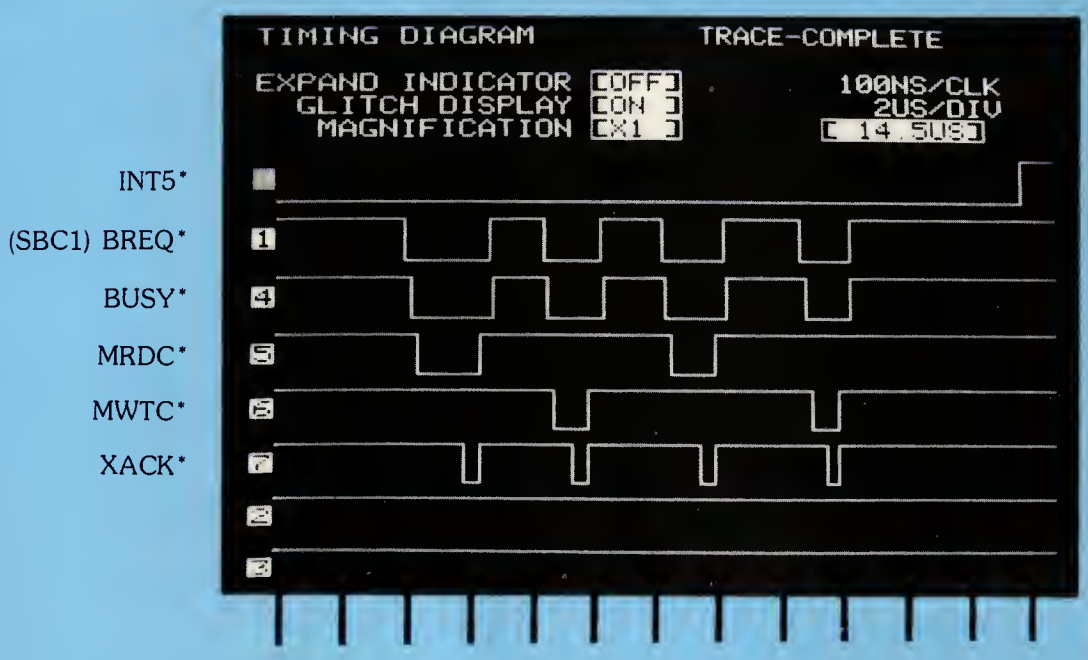


Figure 2. Timing of SBC1's reporting process with no mutual exclusion mechanism and no bus contention.

side shows the high-to-low transition of the interrupt signal issued by SBC0 and the responses of the three PEs. SBC1 responds first, followed by SBC3 and SBC2. The sequence in which the PEs respond to the interrupt is unpredictable; it depends on the state of each processor when the interrupt occurs. Note that the iSBC 86/12A acquires control of and locks the Multibus during its response to an interrupt. This accounts for the three intervals at the start of the timing diagram during which the BUSY* signal is active.

Because the three PEs perform identical processing tasks and execute at about the same rate, the sequence in which the PEs complete their responses to the interrupt is also the sequence in which they complete their processing tasks and attempt to initiate their reporting processes. Table 1 interprets the timing diagram by identifying the bus master, the operation performed, and the value read from or written to DONE or LOC*x* during each Multibus cycle. The table shows, for example, that SBC1 begins reporting first and completes one Multibus cycle (i.e., it reads DONE) before SBC3 begins reporting. SBC3 controls the second bus cycle as it reads DONE. SBC1 and SBC3 control the third and fourth bus cycles, respectively, as they write incremented values to DONE. SBC2 controls the fifth bus cycle as it performs its initial read of DONE. Note that after the fourth bus cycle the value in DONE is 1 even though both SBC1 and SBC3 have incremented DONE. The reporting process is not performed correctly in this run; Table 2 lists the final values of DONE, LOC1, LOC2, and LOC3 for the runs in Figure 2.

Case 1: Locked bus. In this case a processor completes its task and begins its reporting process. Once it gains control of the bus, it maintains exclusive control until it completes its reporting process. It achieves exclusive access to DONE because no other PE is able to gain control of the bus, something which is required to access DONE.

Figure 2b describes bus activity during a typical run for this case. During the reporting process, the BUSY* signal is asserted three times for relatively long periods. The processors complete their tasks in the sequence SBC2, SBC3, SBC1. Because SBC2 gains control of the bus first, it completes its reporting process first. Both SBC1 and SBC3 request control of the bus while SBC2 is reporting, but their requests are not immediately granted because SBC2 retains control of the bus during its reporting period. When SBC2 finally surrenders control of the bus, SBC3 gains control because it has a higher priority than SBC1 in the parallel priority resolution scheme; SBC3 gains control even if it finishes its task *after* SBC1, as long as both requests are active when SBC2 surrenders control of the bus.

Table 1.
Bus master, operation, and data transferred during each Multibus cycle in run for Case 0 in which no mutual exclusion mechanism is used.

	Multibus operation				
Multibus cycle	First read on DONE	Write on DONE	Second read on DONE	Write to LOC*x*	Data on Multibus
1	SBC1				0
2	SBC3				0
3		SBC1			1
4		SBC3			1
5	SBC2				1
6			SBC3		1
7		SBC2			2
8			SBC1		2
9				SBC3	1
10			SBC2		2
11				SBC1	2
12				SBC2	2

Table 2.
Final values of reported variables for runs presented in Figure 2.

	Final value of variable			
Case	DONE	LOC1	LOC2	LOC3
0 No mutual exclusion	2	2	2	1
1 Locked bus	3	3	1	2
2 Flag	3	1	2	3
3 FIFO semaphore	3	1	3	2

Case 2: Flag. A flag is a binary variable which controls access to a resource. If the flag is in its RESET state, the resource is free; if the flag is in its SET state, the resource is busy. Initially, the flag is reset. A processor gains access to the resource by performing a test-and-set operation which tests the state of the flag and then forces the flag to its SET state. If the test indicates the SET state, the processor repeats the test-and-set operation. If the test indicates the RESET state, the processor takes control of the resource and maintains control until it resets the flag. To work correctly, the test-and-set operation must be indivisible; that is, the processor performing the operation must have exclusive access to the flag during the test-and-set operation. In this case access to DONE is controlled by a flag (a location labeled FLAG) located in the dual-port memory of SBC0. Each PE locks the Multibus during its test-and-set operation to ensure mutually exclusive access to FLAG.

Figure 2c describes bus activity during a typical run. The PEs complete their tasks in the sequence SBC1, SBC2, SBC3. On its first test-and-set operation, SBC1 gains access to DONE and begins its reporting process. SBC's bus request signal indicates that SBC1 makes six bus requests during its reporting process. The first is for the test-and-set operation, which requires two bus cycles during which the bus is locked. The next four are those required by the reporting procedure. The last bus request—which requires one cycle—is made so FLAG can be cleared. When SBC2 and SBC3 complete their tasks, the flag is set, causing them to enter loops in which they repeatedly test and set FLAG, generating the regular patterns of bus requests shown in the figure. After SBC1 clears FLAG, SBC2 gains access to DONE because it performs the first test-and-set operation after SBC1 clears FLAG. SBC3 continues to perform test-and-set operations while SBC2 reports and finally gains access to DONE on its eighth try.

Case 3: FIFO semaphore. A semaphore comprises an integer variable (count) and a list of processes or tasks waiting for the semaphore. Two primitive operations, P and V, are defined on the semaphore:

- *P operation.* A process decrements the count of the semaphore. If the count is non-negative, the process continues executing. If the count is negative, the process is blocked and a pointer to the process is added to the list associated with the semaphore.
- *V operation.* This the inverse of the P operation. A process increments the count by 1. If the result is positive, no further action is taken. If it is negative or zero, there is at least one process waiting on the list. One process is removed from the list and reactivated.

Note that the V operation does not specify how the process to be activated is selected; a first-in/first-out scheme, a priority scheme, or some other scheme can be used. The initial value of the count depends on the intended application.

In this case access to DONE is controlled by a semaphore; blocked processes are reactivated on a first-in/first-out basis. The count associated with the semaphore, and a bit map identifying processors waiting on the semaphore, are held in a location labeled FIFO in the dual-port memory of SBC0. The discussion of the reporting process (see page 40) includes a program listing of the FIFO semaphore implementation. In implementing the semaphore, we found it convenient to offset the count value used in the definitions of P and V by -1. SBC0 initializes the count to 0 and clears the bit map. When a PE completes its task and is ready to report, it asserts its bit in the bit map, decrements the count, and copies the resulting value to a local register. This is done in an indivisible operation which is implemented by locking the bus during the operation. The count value copied by the PE determines the PE's position in the list of waiting PEs. If the count is -1, the waiting list is empty; in this case the PE begins its reporting process immediately. If the count is less than -1, the PE computes its position on the waiting list and enters a WAIT state in which it remains until it receives a wake-up call. When a PE completes its reporting process, it negates its bit in the bit map and increments the count in an indivisible operation. If the resulting count value is negative, some PE is waiting to gain access to DONE, and the PE that has just finished reporting issues

wake-up calls to the PEs whose bits are asserted in the bit map. If the resulting count value is zero, the waiting list is empty. When a waiting PE receives a wake-up call, it begins its reporting process immediately if it is at the top of the waiting list; otherwise, it moves itself up one position on the waiting list and reenters the WAIT state. Note that maintenance of the waiting list is distributed among the processors—SBC0 holds the count and bit map identifying the PEs on the waiting list, and each PE keeps track of its position on the list.

The WAIT state and the wake-up call are implemented with the 8086 microprocessor's WAIT instruction.[4,5] This instruction causes the 8086 to enter an idle state if the signal on its TEST* pin is not asserted. The 8086 leaves the idle state when the TEST* signal is asserted. In executing the V operation, a PE updates the FIFO and, if any PEs are on the waiting list, issues an active-low pulse to the TEST* input of the 8086 on each PE waiting on the semaphore. The pulses are issued by toggling on and off corresponding bits in the parallel I/O port. The pulses cause waiting processors to leave the idle state and resume program execution.

Figure 2d describes bus activity during a typical run for this case. The PEs complete their tasks in the sequence SBC1, SBC2, SBC3. SBC1 accesses FIFO first and begins its reporting process immediately. SBC3 accesses FIFO next, places itself at the top of the waiting list, and then enters a WAIT state. SBC2 accesses FIFO last, places itself on the waiting list behind SBC3, and enters a WAIT state. SBC1's bus request signal indicates that SBC1 gains control of the bus six times during its reporting process. The first is the P operation, which requires two bus cycles during which the bus is locked. The next four are those required by the reporting procedure. The last is the V operation, which also requires two bus cycles during which the bus is locked. Following the V operation, SBC1 issues pulses (not shown in the timing diagram) which wake up SBC3 and SBC2. SBC3 exits its WAIT state and begins its reporting process. SBC2 exits its WAIT state, moves itself to the top of the waiting list, and again enters a WAIT state; none of SBC2's activity following wake-up is reflected on the timing diagram because it does not involve the Multibus. Following its reporting process, SBC3 issues a pulse which wakes up SBC2. SBC2 exits its WAIT state and begins its reporting process. No wake-up pulses are issued following SBC2's reporting process.

The results of our study are summarized in Table 3. The locked bus approach permits the reporting process to be carried out in the shortest period, but it has the drawback that it ties up more system resources

Table 3.
Comparison of the three mechanisms based on the cases studied.

Issue	Locked bus	Flag	FIFO semaphore
Typical duration of reporting period for 3 PEs	70 μs	140 μs	155 μs
Number of Multibus cycles to acquire shared resource	0	2 to ∞	2
Number of Multibus cycles to release shared resource	0	1	3
Is Multibus available to other processes during reporting?	No	Yes	Yes
Is Multibus free of contention during reporting process?	No	No	Yes
Number of dedicated I/O bits on each PE when *n* PEs share bus	1	0	*n* + 1
Number of PEs which can share a resource when *n* PEs share bus	*n*	Limited by bus saturation to 3	*n*

than necessary. The resource to which mutually exclusive access is required is the location labeled DONE—the locked bus approach provides mutually exclusive access to the Multibus and effectively to all resources which are accessed via the Multibus. This prevents all other bus masters from using the Multibus during a PE's reporting process. In a system in which other processes (e.g., input/output processes) execute in parallel with the reporting process, it might not be acceptable to tie up the bus for the extended period required in this approach. If additional PEs are included in the test, the total reporting period increases linearly with the number of PEs.

The FIFO semaphore mechanism should produce faster operation and less bus activity than a lock implementation.

The flag mechanism is a software approach which requires additional instructions in the program to manipulate the flag. The overhead arising from these instructions slows the reporting process. However, the greatest drawback of this approach is the increased contention for control of the bus resulting from busy-waiting activity[1,6] caused by repeated test-and-set operations of PEs waiting to gain access to shared memory. One can see this in Figure 2c if one compares the interval required by SBC1 to report (from the end of the test-and-set operation to the end of the clear flag operation) to the interval required by SBC3. During SBC1's reporting process, the BUSY* signal indicates the bus is busy almost continuously. SBC1's reporting process takes nearly twice as long as that of SBC3. This difference is due to the heavy demand for use of the bus caused by the busy-waiting activity of SBC2 and SBC3 during SBC1's reporting interval. There is no contention for the bus during SBC3's reporting interval.

In contrast to the locked bus, the flag mechanism does not prevent other masters from using the Multibus during a PE's reporting process. However, if additional PEs are included in the test, a situation can arise in which the reporting task cannot be completed. This can occur if a low-priority PE gains access to the shared data but is subsequently unable to gain control of the Multibus because the Multibus becomes saturated by the busy-waiting activity of higher-priority PEs. In fact, deadlocks of this type occur when a fourth PE is included in the tests.

The semaphore approach is slower than the flag approach for the cases in Figure 2 because more instructions are needed to manipulate the semaphore than to manipulate the flag. In the absence of contention, a PE executes three instructions to test and set the flag and initiate the reporting process; one instruction resets the flag. The instruction which copies and sets the flag includes a LOCK prefix; the bus is not locked while the flag is being reset. Under the same conditions, the P operation requires ten instructions and the V operation six instructions. Both operations include two input/output instructions to lock the bus during semaphore manipulation.

In the semaphore approach there is no busy-waiting activity creating contention for control of the Multibus during the reporting process. There is some contention for control of the Multibus when the three PEs try to decrement the semaphore at the same time. Once these initial operations have been completed, however, the PEs do not contend for control of the Multibus; the reporting PE reports without interference from other PEs. Processes operating in parallel with the reporting task can use the bus concurrently without encountering abnormal bus contention due to the reporting process. If additional PEs are included in the test, the reporting period increases linearly with the number of PEs, if the effects of initial contention for the semaphore are ignored.

The FIFO semaphore mechanism is similar to the lock mechanism of the C.mmp system.[7] The C.mmp lock provides a centralized waiting list (bit map) and uses separate interrupt lines to each PE as wake-up signals. If a PE is blocked when it attempts to lock a resource, it sets its bit in the bit map associated with the resource and enters a wait-for-interrupt state. When the PE controlling the resource unlocks (releases) it, it issues interrupt signals to the PEs whose bits are set in the bit map for the resource. The blocked

PEs, upon receipt of the interrupt, resume execution and contend for the resource; one, determined randomly, gains control of it.

The FIFO semaphore mechanism and the C.mmp lock differ primarily in the way a PE responds to a wake-up signal. The FIFO semaphore mechanism selects a PE from the waiting list using a first-in/first-out policy. The lock mechanism implements a random selection policy, though it could support other selection policies at a cost of additional processing time. By distributing maintenance of the waiting list among the waiting PEs, the FIFO semaphore mechanism eliminates the bus activity generated by waiting PEs when a resource is released in a system with a C.mmp-type lock. In tests of the type described here, the FIFO semaphore mechanism should produce faster operation and less bus activity than a lock implementation.

Acknowledgments

This work was supported by the National Aeronautics and Space Administration under Grant NAG 3-113 to Arizona State University. We thank Shih-Hung Lin for assistance in the final test runs.

References

1. P. Brinch Hansen, *Operating System Principles,* Prentice-Hall, Englewood Cliffs, NJ, 1973.
2. B. A. Bowen and R. J. A. Buhr, *The Logical Design of Multiple-Microprocessor Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1980.
3. E. P. O'Grady, "A Communication Mechanism for Multiprocessor Simulation Systems," *Simulation,* Vol. 34, No. 2, Feb. 1980, pp. 39-49.
4. R. Rector and G. Alexy, *The 8086 Book,* Osborne/McGraw-Hill, Berkeley, CA, 1980.
5. A. C. Hartmann and S. Fehr, "A VLSI Architecture for Software Structure: The Intel 8086," *IEEE Micro,* Vol. 1, No. 2, May 1981, pp. 57-69.
6. K. Muhlemann, "Busy Waiting in Multiple-Processor Synchronization," *IEE Proc.,* Vol. 127, Pt. E, No. 3, May 1980, pp. 85-87.
7. H. M. Mashburn, "The C.mmp/Hydra Project: An Architectural Review," Chapter 22 in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples,* McGraw-Hill, New York, 1982.

E. Pearse O'Grady is an associate professor in the Department of Computer Science at Arizona State University. He will be a Fulbright lecturer in Ireland at University College Cork for the 1985-86 academic year. Prior to joining ASU, he was with the University of Maryland and the McDonnell-Douglas Astronautics Company. His research interests are in parallel processing and continuous system simulation. O'Grady received his BS from Saint Louis University and his MS and PhD from the University of Arizona, all in electrical engineering.

Raul Lozano is a design engineer at Data/Ware Development, Inc., in San Diego. He was previously with the Rockwell International Avionics and Missiles Group. His current interests include fault-tolerant hardware design in multiprocessor computer systems. Lozano received his BS from the University of Wisconsin and his MS from Arizona State University, both in electrical engineering.

Questions regarding this article can be directed to E. Pearse O'Grady, Department of Computer Science, Arizona State University, Tempe, AZ 85287.

A Real-Time Fortran Executive

Daniel A. Crowl
Wayne State University

This executive provides its users with sophisticated real-time functions without requiring them to do a lot of complicated assembly language interfacing.

Microcomputers today are readily equated with word processing, database management, accounting, and spreadsheets. These uses are considered the classical application domain of microcomputers. However, the saturation limit for microcomputer software and hardware applications is far from being reached. The potential for applications in other, nonclassical areas still appears almost limitless.

A growing area of microcomputer application is in real-time processing. Unfortunately, "real-time processing" is difficult to explain and is one of those phrases that everyone knows the meaning of but can't provide an adequate definition for. For the purposes of this article, real-time processing is an operation performed by a computer that involves a well-timed interaction with the physical world. A classical example appears in the laboratory, where a microcomputer may be employed for data acquisition and experiment control. In this case, the microcomputer is required to collect data over very well specified time intervals and is also required to adjust or change the experiment to user specification. Thus, the microcomputer is forced into a "real-time environment" where the normal pace of the physical world dictates the speed of events. A much more detailed discussion of real-time processing is provided by Mellichamp.[1]

Unfortunately, real-time processing requires rather sophisticated programming to ensure that the computer is available for service when a particular real-world event occurs. A single dedicated terminal can be scanned repeatedly within a software loop to check for any activity. But increase the number of terminals to dozens and this polling technique can lead to missed data on some terminals when the computer is busy elsewhere.

The solution to the real-time processing problem is embodied within a real-time operating system. Such an operating system must be installable on a user's system and

(1) must support a variety of real-time functions,
(2) must be interfaceable to a higher-level language for ease of use,
(3) must be transportable and readily implementable, involving a minimum of tedious assembly language programming, and
(4) must support multiple-task execution.

The second requirement is essential for rapid development of real-time applications software, especially applications programs written by people with minimal programming experience. The third requirement ensures maximum utilization of the operating system on a wide variety of computers with any configura-

0272-1732/85/0800-0048$01.00 © 1985 IEEE

tion of available peripherals. The last requirement must be met if the executive is to have any useful capabilities at all. More about this later.

Most real-time operating systems available today fail to meet one or more of these requirements. In most cases the operating system is designed for a particular computer system having a specific set of peripherals or requiring extensive assembly language programming for peripheral interfacing or applications programming.[2]

The real-time operating system to be described in this article is not a true operating system since it does not replace the operating system on the user's computer. Rather, it is designed to appear as an application program to the existing operating system. For this reason, it is called a real-time executive system. The advantage to this approach is that the user can use his previously developed hardware interfacing—i.e., his terminal, floppy disks, and so forth. The disadvantage is that a loss of efficiency can occur, particularly with polled I/O systems.

The requirements discussed previously are all met by EXEC-80, the real-time executive system to be detailed here. It was designed to interface to an existing, unmodified CP/M operating system supporting a Fortran IV compiler. However, because its functions are not operating-system- or Fortran-compiler-dependent, it can be implemented on a variety of other systems. Although EXEC-80's coding is in 8080A assembly language, its fundamental concepts can be applied to practically any computer and operating system.

This article is designed to provide a background on real-time executive function and to detail how a specific real-time executive was implemented on existing hardware and software. While the programming is much too extensive to allow comprehensive listings to be included here, critical concepts are explained and illustrated.

Real-time processing—an analogy

The operation of a telephone answering service is a practical example of real-time processing. Consider an operator whose duty is to take messages from four different telephones. Each telephone has four different states: inactive, ringing, holding, or active. The telephones are arranged on a priority basis, with telephone one having the highest priority. Priority here means that if a call comes across telephone one while the operator is taking a message on telephone two, then telephone two must be immediately placed on hold so telephone one can be answered. And if a call comes across telephone two while a message is being received on telephone one, telephone two cannot be answered until the message from telephone one is completely received. We will assume that the callers are aware of the priority system and will let the telephone keep ringing if it is not immediately answered.

If calls are few and widely spaced, all telephones can be answered immediately, and all messages can normally be taken before the next call is received. However, as message frequency increases, conflicts will arise and the priority system must be invoked. When this occurs, lower-priority callers become aware of longer delays before the phone is answered. They also notice an increase in the number of times they are placed on hold. If messages are quite numerous, the lower-priority callers can be left ringing since the operator may have to spend all his time answering the higher-priority telephones.

Obviously, in this configuration the calls that are deemed the most important must be the ones given the highest priority. The assigning of priority to messages as well as to lines is critical to the efficient operation of the system. Those with the highest-priority messages must be given sole access to the highest-priority lines. It would make little sense to have a system that prioritized incoming lines but not the messages received on them.

A situation can develop in which the priority system must be revoked. This occurs when conflicts over resources common to all telephones arise. Suppose our system supports a single telecopier that is shared by all the telephones. Let us assume that a transmission over the telecopier cannot be interrupted. Once the unit is connected to a particular telephone, the transmission must be completed before the

unit can be moved to another telephone. Hence, if a low-priority caller is using the telecopier, a higher-priority caller must wait until the transmission is complete before he can send his message to the telecopier.

A software analogy

Our telephone message system works in a fashion identical to that of a real-time executive with many tasks to perform. A "task" is the software equivalent of an individual telephone unit. It can be defined as a "logically complete program segment."[2] Just as in the telephone message system, software tasks in a computer system are assigned priorities, with activity within the tasks dependent on who requests service and who has the highest priority. The assignment of these tasks and their respective priorities is entirely application-dependent. The ultimate efficiency of the resulting system is highly dependent on this structure.

The reasons for adopting the task structure are numerous. First, it provides enormous flexibility in separating operations that have different time frames. For instance, hourly operations can be assigned a task different from that assigned to operations that must be performed each minute. Second, it allows operations that differ in function to be separated. Data collection operations can be assigned a task different from that assigned to plotting operations. Finally, it enables priority to be assigned, with perhaps data collection or control being given the highest priority and on-line plotting the lowest.

Tasks can have four states: off, bid or running, suspended, or time-delayed. These states have analogs in the telephone example. OFF represents a state of no activity. A task that is OFF does not request service by the CPU. BID is equivalent to a ringing telephone. It is a request for CPU service. SUSPENDED is identical to an indefinite "hold" on the telephone; task execution has been stopped somewhere and is to be restarted at a later time. TIME-DELAYED is the same as a telephone hold requested by the caller, although a TIME-DELAYED state lasts for a fixed period of time. This is the most powerful utility of the real-time executive system, providing the only link to the real-time world. Finally, RUNNING means that the task is active and requesting as much CPU service as is available within the constraints of the priority system. Note that RUNNING is identical to a BID, since the task is requesting as much CPU as is available.

All of the tasks within the system must share common system resources such as the floppy disk drive, system console, and plotter. These devices are analogous to the telecopier unit in the telephone example. If a task is using the printer, for example, another task cannot be allowed to use it until the first task is completed. Otherwise, printed messages from several tasks may become intermixed and scrambled. Common system resources are not necessarily hardware devices, either: they can be software resources like a system or utility subroutine.

Let's return to the laboratory example cited earlier. In this example, the highest-priority task is data collection: the data must be collected over precise time intervals. This operation requires minimal CPU service. The second-highest-priority task is the control function, since a well-controlled experiment is desirable. The remaining tasks are divided among data storage, data processing, and on-line data plotting. These functions are not time-critical and are included within the system for convenience.

Real-time clocks and interrupts

The real-time executive must repeatedly review the system status to determine if a higher-priority task is requesting service. This operation should be performed during rather short and well-defined time intervals. A real-time clock and a processor-supported interrupt system are essential for this function. Without either of these hardware items, the real-time executive will not be capable of regaining CPU control once it has been released to a task.

The function of the real-time clock is to continuously generate precisely timed interrupts. This is unlike the interval timer on many other systems in which the interval must

be reset by software after each interrupt; a real-time clock should generate continuous, unattended interrupts.

A task can be interrupted at any point in its execution sequence. However, the computed results of the task must be independent of when and where the task was interrupted. It is the function of the executive to ensure that all transient information related to the particular task is saved so that it can be restored before the task is restarted. Therefore, the real-time executive must save and restore all registers, stack pointers, and system flags, and any system memory locations shared in common by the tasks.

The time interval selected for the real-time clock interrupt depends, of course, on what hardware is available. If the interval is too short, system efficiency declines, since the executive uses most of the CPU time and leaves little for individual tasks. If the interval is too long, high-priority tasks are unable to gain CPU control fast enough, with the result that important real-time events are missed. Our experience shows that 0.02- to 0.1-second intervals are appropriate for most applications.

Interfacing to an existing high-level language

Most compilers generate minimal applications code and depend heavily on run-time libraries for execution of different types of statements. Most of the user code generated is simply subroutine calls to this library for DO-LOOP processing, IF statement processing, numerical processing, and so forth.

Suppose that for a particular application a user generates four subroutines corresponding to four different tasks. The high-level language (let's assume Fortran) compiles the source code and produces an object deck. Unfortunately, all four tasks have calls to common run-time subroutines. Thus, when all programs are simultaneously loaded into memory by a linking loader, only one copy of the run-time subroutine is loaded, with branch addresses in all four tasks resolved to the same subroutine. The problem that can arise is shown in Figure 1. Suppose Task 2 calls one of the run-time subroutines shared by Task 1. The run-time subroutine uses a special data area for intermediate result computation. However, during the course of its calculation, an interrupt occurs, with subsequent processing by the real-time executive. The real-time executive determines that Task 1 has a higher priority and transfers the CPU. Task 1 calls the common run-time subroutine and changes the data stored in the run-time subroutine's intermediate data area! The intermediate data value needed for the correct completion of Task 2's computation has been altered. When Task 1 is completed, the executive returns the CPU to Task 2, but Task 2's computation has been upset.

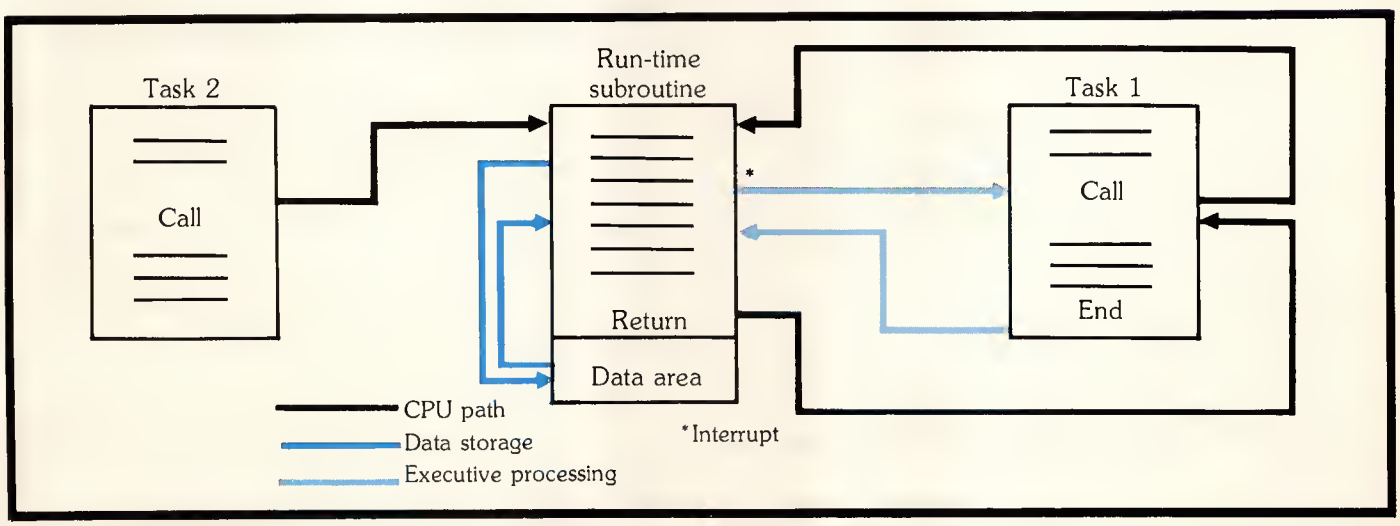


Figure 1. Reentrancy problem with shared run-time subroutines.

Since registers can be saved and restored quite easily, subroutines that use only registers can be interrupted and rerun any number of times without being upset. These special types of subroutines are called reentrant.

EXEC-80 handles the problem of nonreentrancy of the run-time subroutines in a special fashion. If the high-level language is PROM-burnable, then the data storage areas must be separable from the nonmodifiable code areas. Otherwise, separate PROM and RAM areas cannot be assigned. If the modifiable data area associated with the run-time subroutines can be identified, then these areas can be saved as tasks are switched by the executive. This is the approach used by EXEC-80; it is shown in Figure 2. Each task has associated with it a data save area that is used to save the run-time subroutine library data area as the CPU is switched between tasks. The real-time executive is responsible for saving and restoring this data area. While this approach is somewhat inefficient, since the data area is repeatedly swapped, experience shows that only a few hundred bytes need to be moved. Furthermore, this approach has the great advantage that the original, unmodified high-level language can be used directly within the real-time environment once the critical data areas have been identified.

Practical experience shows that not all of the data areas need to be saved. Since the I/O processing routines and I/O buffers should only be used by one task at a time (a case analogous to the telecopier example cited earlier), these data areas do not need to be saved if the user carefully controls access to them. Only areas associated with floating-point computations and DO loop and other internal processing must be saved.

The individual tasks themselves do not need to be reentrant since their data areas cannot be accessed by the other tasks. Data can be transferred between tasks by use of COMMON and other such shared memory techniques.

One must also provide the correct calling protocols between the real-time executive routines (written in 8080A assembler) and the high-level language. Differences in these protocols can be accommodated by personality module interfacing between the high-level language and the executive. EXEC-80 directly supports Microsoft Fortran calling protocols.

The applications program structure for interfacing to a high-level language is shown in Figure 3. Each task is supplied in the form of a subroutine without arguments. A once-called main program must also be supplied by the user. The function of the main program is to

- acquire initial execution from the operating system,

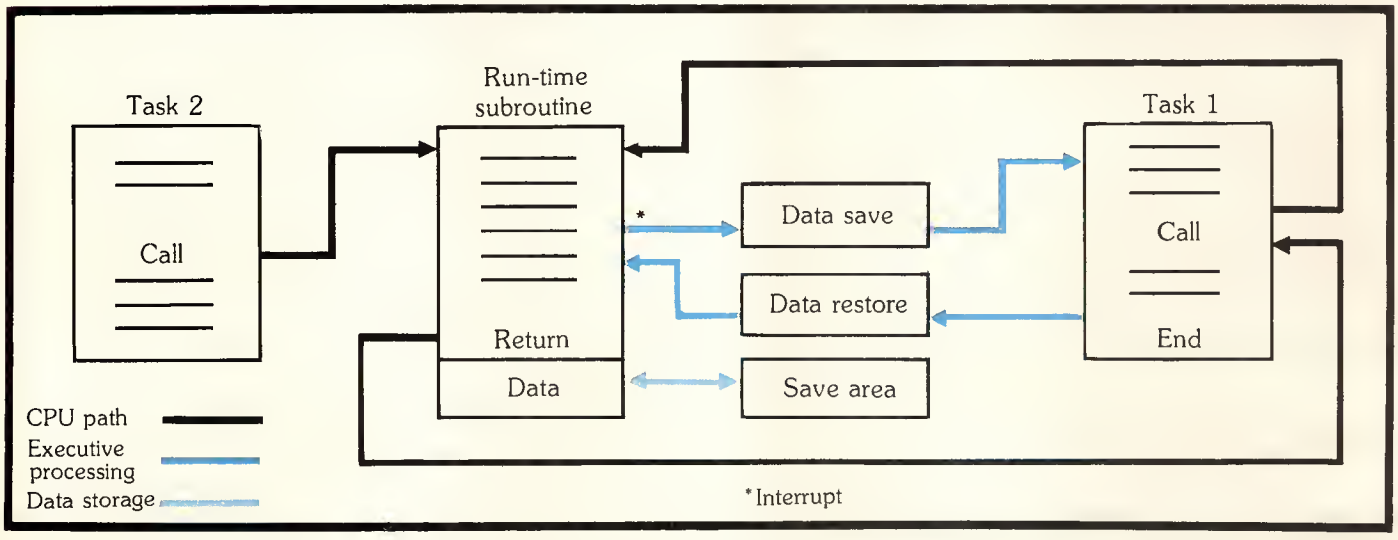


Figure 2. Solution to the nonreentrant run-time subroutine problem.

• perform any user-required hardware and software initialization,
• specify the addresses of the task subroutines to the real-time executive, and
• call the executive.

The last executable statement in the main program is a call to a subroutine called TASKS. This subroutine is the initial entry point to the executive; the subroutine call contains the addresses of the individual tasks as arguments.

Table 1 contains a listing of the user-callable subroutines supported by the executive. Notice that the sole real-time control function is the time delay routine called TMDL. This routine can be used to delay execution of the calling task by any number of clock counts. Other real-time functions (such as execution of a task at a specified time)[3] can be built up by the user from this primitive function. Inclusion of these more sophisticated functions reduces the installation-independence of the executive; they can all be implemented in the high-level language.

Two essential routines, BSY and NOTBSY, are listed in Table 1. These routines are used to control the allocation of system resources. Here "resources" can mean either I/O resources, such as the system console or printer, *or* software resources, such as nonreentrant subroutines shared by different tasks. The user must remember that these routines provide only a software mechanism for controlling system resources. Their function is only to maintain internal executive tables on the usage of these resources.

Suppose that within an applications program task an output on the system console is required. Since other tasks may occasionally use the console, a call to executive subroutine BSY must be made before the output is initiated. A single, arbitrary argument passed to BSY indicates the particular device. Subroutine BSY checks a table for the status of the selected device. If the device has been flagged as busy, then the calling task is repeatedly time-delayed until the device is available. If the device is not busy, then it is flagged as busy and task execution continues. Subroutine NOTBSY is called by the task after it has finished with the device. This routine flags the device as available. It is the responsibility of the user to

Table 1.
Subroutine functions provided by EXEC-80.

Subroutine	Function
TASKS	Provides entry point to executive from once-executed, user-supplied main program.
TMDL	Delays calling task by specified number of clock ticks.
BID	Bids a task.
SUSPND	Suspends calling task.
UNSPND	Unsuspends specified task.
BSY	Requests use of a device.
NOTBSY	Declares device not in use.
DVSTAT	Returns device status.
INTON	Turns interrupts ON.
INTOFF	Turns interrupts OFF.
STOP	Terminates calling task.

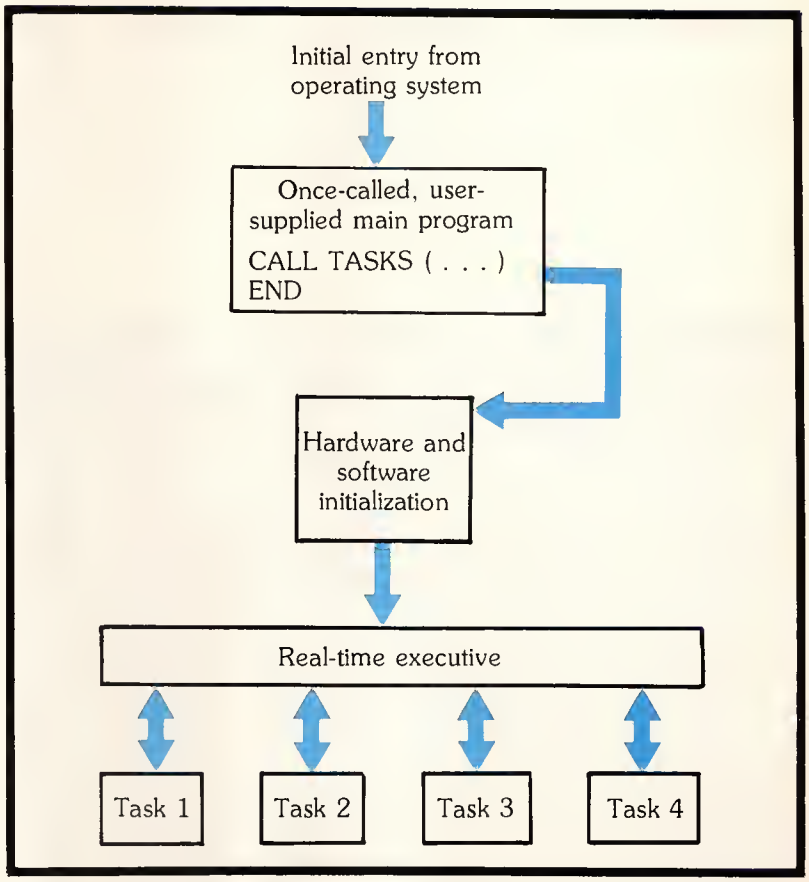


Figure 3. Initial entry to executive.

ensure that NOTBSY is called when the use of a system resource has been completed. Failure to call NOTBSY can result in the other tasks being permanently time-delayed.

Interfacing to the operating system

My objective was to develop a real-time executive that requires little or no modification of the existing microcomputer operating system. Two particular cases must be considered: polled and interrupt-driven environments.

The major function of the operating system is to provide I/O interfacing. In the polled mode of I/O, the operating system constantly checks I/O devices to determine if output has been completed or if input data are available. In the interrupt mode, completion of data output or input generates a CPU interrupt, forcing the processor to jump to an interrupt-processing routine. Obviously, the interrupt mode of operation is the most efficient, since the processor can tend to other devices while waiting for I/O on a device to be completed. The polled mode is much easier to program and is the usual method employed in microcomputers dedicated to single users. Both the polled mode and the interrupt-driven mode can be handled by EXEC-80.

In both cases, access to the operating system means, in almost all situations, that a particular I/O operation is requested. I/O events must be carefully controlled by the user (with routines BSY and NOTBSY) to ensure that only one task is performing a particular I/O function at any one time. In this way, possible conflicts within the operating system due to nonreentrant I/O requests are eliminated.

In the polled mode of operation, the operating system continually checks the status of the system devices to determine if I/O has been completed or is being initiated. This type of system functions without interrupts and is not affected by the real-time clock interrupts initiated by the executive.

In an interrupt-driven system, the real-time executive and operating system must function in the same interrupt environment. Conflicts can arise if careful consideration is not given to both environments. Many interrupt-driven operating systems provide a real-time clock jump function that can be used to route clock interrupts to any user-specified routine. In the case we are discussing, this jump should be routed to the executive clock interrupt entry point. Operating systems that do not provide this utility must be modified in some other fashion.

The question of interrupt priority can be important for interrupt-driven systems. The real-time executive functions best when the real-time clock is given the highest priority. However, experience on systems with lower real-time clock priorities has indicated no noticeable loss of executive function.

Table 2.
Entry points and functions of user-supplied XIOS.

Entry	Function
XSAVE	Specifies data areas in user's higher-level language that must be saved.
XINIT	Initializes user's interrupt hardware and software.
XSWITCH	Returns switch pattern from user's switch hardware.
XLIGHT*	Outputs task status pattern to user's light hardware.
XTIME*	User-supplied timekeeping function.
XACK*	Hardware interrupt acknowledge.
XIDLE*	Idle routine.

*These routines must have an entry point but can minimally consist of a RET instruction.

Interfacing to a particular installation

To provide maximum flexibility in using the real-time executive with a host of configurations, some installation-dependent routines must be supplied by the user. These routines should be kept to a minimum to improve utilization.

All installation-dependent routines are contained within an XIOS (executive I/O subroutine). The user-supplied functions con-

tained in the XIOS are listed in Table 2. Note that there are both hardware- and software-dependent parts of the XIOS. The interrupt initialization routine and the acknowledge, user switch, and light routines in the XIOS are user-hardware-dependent, while the data-area-save, timekeeping, and idle functions are software-dependent.

A sample XIOS implementing the minimal functions is shown in Figure 4. The linking conventions are those supported by the Microsoft linking loader, L80, while the

```
;
;       Skeletal executive XIOS
;
;       This XIOS includes the following entry points:
;
;       1. XINIT—entry for initialization
;       2. XSWITCH—entry for user's switch reading
;       3. XLIGHT—entry to display lights
;       4. XTIME—user's timekeeping routine
;       5. XACK—user's hardware-dependent interrupt processing
;       6. XIDLE—idle routine
;
;       Information on data storage areas is also provided to the Executive
;       via entry point XSAVE.
;
        ENTRY       XSAVE,XINIT,XSWITCH
        ENTRY       XLIGHT,XTIME,XACK,XIDLE
;
;       External reference to executive interrupt entry point.
;
        EXT   XINT
;
;       External reference to data area in Microsoft
;       Fortran V3.44.
;
        EXT   $$A1,$DTBF
;
;       Executive save area information block.
;       First block contains addresses to individual task save areas.
;
            DSEG
XSAVE:      DW      AREA1   ;Task 1
            DW      AREA2   ;Task 2
            DW      AREA3   ;Task 3
            DW      AREA4   ;Task 4
;
;       Information on areas to save in Microsoft Fortran
;       V3.44.
;
            DB      124     ;124 bytes to store.
            DW      $$A1    ;Base entry point of store area in module FORDAT
```

cont'd next page

Figure 4. Sample XIOS.

assembler language conforms to Microsoft's M80. The XIOS must be written in assembler, since this is the only easy way to provide the data save information.

The first function of the XIOS is to interface to the user's real-time clock hardware. This function is called only once and is performed by the XINIT entry point in the XIOS. Routine XINIT must initialize the user's hardware and set up the correct processor interrupt jump vectors. For the sample XIOS shown, only the simplest interrupt

```
                                    ;   of Fortran library.
                DB      149         ;149 bytes to store.
                DW      $DTBF       ;Entry for module DTBF.
                DB      0           ;End of storage commands.
;
;
;       Storage areas for each task.
;
AREA1:          DS      273         ;Task 1
AREA2:          DS      273         ;Task 2
AREA3:          DS      273         ;Task 3
AREA4:          DS      273         ;Task 4
;
;       Data area for idle count.
;
DATA:           DW      0
;
                CSEG
;
;
;       XINIT entry point.
;
;       This routine is called *once* by EXEC-80. It performs
;
;       1. user's interrupt hardware initialization
;       2. user's interrupt software initialization
;
;       More sophisticated programmers might want to establish a jump table for using the
;       more powerful interrupt capabilities of the Z80.
;
XINIT:
                MVI     A,0C3H      ;Get branch instruction.
                STA     038H        ;Lay down at location 38H.
                LXI     H,XINT      ;Get executive interrupt address.
                SHLD    039H        ;Lay down at location 39H.
;
;       Begin hardware initialization.
;
;       User's hardware initialization goes here.
;
                RET                 ;Return to EXEC-80. This routine does not issue EI
                                    ;   instruction.
```

mechanism—with an interrupt jump to location 38 hex—is implemented. More sophisticated interrupt mechanisms, particularly the more sophisticated Z80 types, can be developed if required.

Some interrupt systems, particularly those with priority interrupt controllers, require an end-of-interrupt acknowledge after each interrupt. This function is provided by routine XACK. The sample XIOS does not require this function and merely returns to the calling executive.

```
;
;       Entry point for user's switch-reading routine.
;
;       This version simply turns all tasks ON.
;
XSWITCH:
                MVI     A,0FH       ;Turn all four tasks ON.
                RET                 ;Return.
;
;       Entry point for user's light routine.
;
XLIGHT:
;
;       User's output software for lights goes here.
;
                RET                 ;Return to EXEC-80.
;
;       Entry point for user's timekeeping routine.
;
;       No timekeeping supported in this version!
;
XTIME:
                RET                 ;Return.
;
;       Entry point for user's hardware-specific interrupt-acknowledging routine.
;
XACK:
;
;       User's hardware-dependent interrupt-acknowledge software goes here.
;
                RET                 ;Return to EXEC-80.
;
;       Entry point for user's idle routine.
;
;       This version counts!
;
XIDLE:
                LHLD    DATA        ;Get current count.
                INX     H           ;Increment.
                SHLD    DATA        ;Save count.
                RET                 ;Return to EXEC-80.
;
;
                END
```

The user switch interfacing is an important option that can be used effectively in a laboratory environment. User switches can be used to control the execution of the various tasks. If four tasks are used, then four switches can be optionally provided by the user, with each switch corresponding to one of the tasks. The executive repeatedly scans these switches. If a switch is turned ON, the executive starts the corresponding task; if it is turned OFF, the executive suspends the corresponding task. This capability gives the user enormous flexibility in controlling laboratory equipment without the system console. The user lights are used in conjunction with the switches: an ON light indicates that the corresponding task is active. Both the switch function and the light function are optional and can be easily overriden by providing a dummy-switch-reading routine in the XIOS. The sample XIOS does not support either switches or lights and simply turns all four tasks ON.

The user switch interfacing is an important option that can be used effectively in a laboratory environment.

A timekeeping option is also provided in the XIOS for those users desiring this capability. This routine (entry point XTIME) is called once for each clock interrupt. The sample XIOS does not implement this feature.

An XIDLE entry point is also provided in case the user wants to utilize any system idle time. Idle time occurs when all tasks are inactive. The user may wish to utilize XIDLE for system housekeeping functions such as updating the time on the system console.

The data areas to be saved from the user's high-level language are specified with the XSAVE entry point. The first four words of

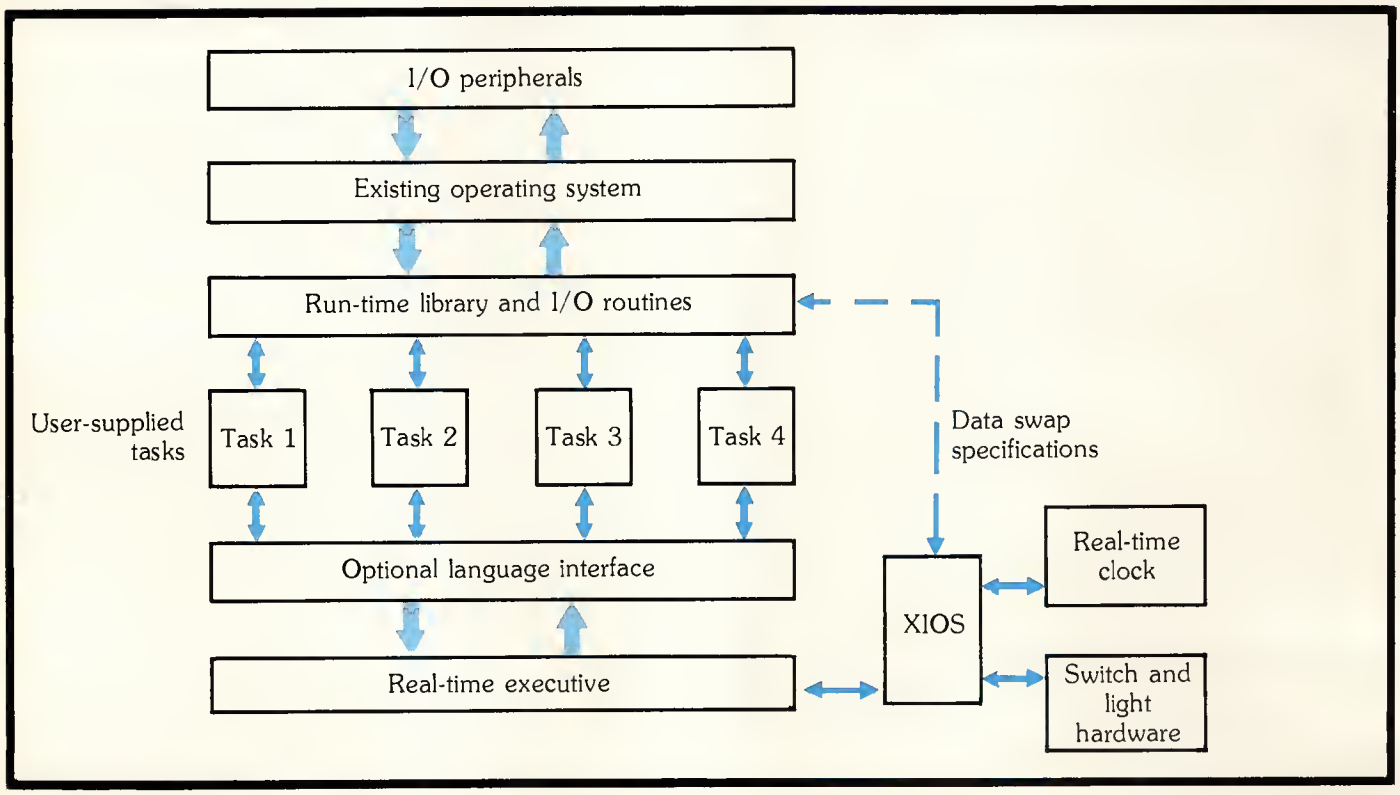


Figure 5. Structure of the complete system.

this specification must contain the addresses of the areas provided in the XIOS for storage. The data blocks following this provide information on the address and length of the areas in the high-level language that are to be saved. Any number of areas can be specified here, with the end of the specification denoted by a "0" in the length field. The areas shown in the sample XIOS are adequate for most applications using Microsoft Fortran Version 3.44.

The complete system

Figure 5 is a conceptual block diagram of the complete system structure. The operating system, Fortran run-time library, and Fortran I/O routines are unchanged from their stock implementation. The real-time executive contains only language- and hardware-independent programming; it remains unchanged as well. The user must supply

- the applications programming,
- the interface between the high-level language and the executive, and
- the hardware interfacing for the executive in the form of an XIOS.

The second and third items have to be developed only once for the particular language and hardware.

The solid lines between the blocks indicate possible lines of execution. The dashed line between the XIOS and the Fortran library conceptualizes the transference of the data save areas.

A sample application program

Figure 6 is a listing of a sample application program written to use the EXEC-80 real-time executive. This application program supports three tasks: a task to collect analog data, a task to save the collected data on floppy disk, and a task to plot the collected data. This program assumes that the user has implemented the user switch control capability discussed earlier.

The function of the once-executed main program is simply to transfer control to the real-time executive and to specify the individual tasks. Note that, by Fortran convention, the names of the tasks are declared by an EXTERNAL statement.

By real-time executive convention, Task 1 has the highest priority while Task 3 has the lowest. The tasks in our sample program have been carefully structured according to this priority. Thus, Task 1 has been assigned to the very important analog data collection activity; Task 2 to the data save operation; and Task 3 to data plotting, an operation that is not time-dependent and is merely provided as a convenience for on-line observation of data. Task 2 could have been combined with Task 1; however, separation of these tasks leads to much improved user control over these operations.

The operating system, Fortran run-time library, and Fortran I/O routines are unchanged from their stock implementations.

Once execution has been passed to the executive through subroutine TASKS, the executive monitors the status of the user switches. To begin the data collection task, Task 1, the user turns Switch 1 ON. This task continues to collect data until either Switch 7 is turned ON or the maximum number of data points (500) is reached. After data collection has been terminated, a new collection sequence can be started by turning Switch 1 OFF and then ON again. The executive recognizes only changes in switch status and will not start or restart a task until a switch has changed status. Note that the statement NPTS = 0 ensures that data collection is reset whenever Task 1 is started or restarted. This facility enables the user to work with independent sets of data or to start over when a false start occurs.

Task 1 collects data about every 60 seconds, as specified to the call to routine TMDL. (The clock interrupt frequency in our example is 0.1 seconds.) It is essential that higher-priority

```
C
C     SAMPLE APPLICATIONS PROGRAM DEMONSTRATING
C     USE OF EXEC-80
C
C     THIS PROGRAM PERFORMS A LABORATORY DATA
C     COLLECTION AND DISPLAY FUNCTION.
C
C     TASK1 - COLLECTS DATA FROM SINGLE ANALOG CHANNEL
C     TASK2 - SAVES COLLECTED DATA ON FLOPPY DISK FILE
C     TASK3 - PROVIDES ON-LINE PLOTTING OF DATA
C
C     THIS PROGRAM ASSUMES THAT USER SWITCHES ARE PROVIDED
C     AND THAT CLOCK INTERVAL IS 0.1 SECOND.
C
C     MAIN PROGRAM
C
      EXTERNAL TASK1, TASK2, TASK3
C
C     ENTER EXECUTIVE
C
      CALL TASKS(3,TASK1,TASK2,TASK3)
      END
C
      SUBROUTINE TASK1
C
C     TASK TO COLLECT ANALOG DATA AT INTERVALS OF
C     60 SECONDS.
C
      COMMON NPTS, V(500)
C
C     ZERO NUMBER OF DATA POINTS
      NPTS = 0
C
C     ENTER DATA COLLECTION LOOP
100   NPTS = NPTS + 1
C
C     CHECK FOR MAXIMUM POINTS
      IF(NPTS.GT.500) CALL STOP
C
C     CHECK FOR STOP SWITCH
C     CALL USER-SUPPLIED SWITCH ROUTINE
      CALL SWITCH(7,J)
      IF(J.EQ.1) CALL STOP
C
C     GET DATA USING USER-SUPPLIED ANALOG ROUTINE
      CALL ANIN(V(NPTS))
C
C     DELAY FOR 60 SECONDS (ASSUMES 10 TICKS PER SEC.)
      CALL TMDL(599)
C
      GO TO 100
      END
C
```

Figure 6. Sample applications program using EXEC-80.

```
      SUBROUTINE TASK2
C
C     SUBROUTINE TO SAVE COLLECTED DATA ON FLOPPY DISK
C
C     FLAG I/O AS BUSY
      CALL BSY(1)
C
C     OPEN FLOPPY DISK FILE
      CALL OPEN(7,'DATA          ',1)
C
C     ENTER DATA SAVE LOOP
      DO 100 I = 1,NPTS
      TIM = FLOAT(I-1)*60.0
      WRITE(7,101) TIM,V(I)
101   FORMAT('  ',2F10.2)
100   CONTINUE
      ENDFILE 7
C
C     DISPLAY CONSOLE MESSAGE
      WRITE(1,102)
102   FORMAT('0', 'DATA SAVE COMPLETED')
C
C     FLAG I/O NOT BUSY
      CALL NOTBSY(1)
C
C     TERMINATE TASK
      CALL STOP
      END
C
      SUBROUTINE TASK3
C
C     TASK TO PLOT DATA USING USER-SUPPLIED PLOTTING
C     ROUTINE "PLOTS" TO PLOT DATA ON SYSTEM CONSOLE.
C
      COMMON NPTS,V(500)
      DIMENSION TIM(500)
C
C     GENERATE TIME VALUES
      DO 100 I = 1,NPTS
100   TIM(I) = FLOAT(I-1)*60.0
C
C     FLAG I/O BUSY
      CALL BSY(1)
C
C     CALL USER PLOT ROUTINE
      CALL PLOTS(NPTS,TIM,V)
C
C     FLAG I/O NOT BUSY
      CALL NOTBSY(1)
C
C     TERMINATE TASK
      CALL STOP
      END
```

tasks call either subroutine TMDL or STOP: a loop in these routines prevents lower-priority tasks from executing.

Task 2 is the routine to save stored data on floppy disk. This operation can be performed while Task 1 is still in operation, but it is probably best performed after data collection has been completed. This task is initiated by turning Switch 2 to ON.

Subroutines BSY and NOTBSY are used within Task 2 to avoid a possible I/O conflict between Tasks 2 and 3. When Task 2 writes to the floppy disk and Task 3 uses the system console, a possible conflict can arise, since both devices use the same Fortran format-processing routines and buffers. Both routines use an arbitrarily selected Device 1 for this purpose.

What is unique about our executive is that it interfaces to a standard operating system running an off-the-shelf Fortran compiler.

Task 3 is the lowest-priority task. It is started by turning Switch 3 to ON, but only executes if both Tasks 2 and 3 are not active. This task can be started at any time and is very useful for graphically displaying the progress of data collection.

Note that all tasks use real-time executive subroutine STOP to terminate task execution. Use of the Fortran-supported STOP statement results in a system boot followed by termination of the program.

Users may also find that interrupts must be turned OFF during floppy disk I/O. Some floppy disk processors use critically timed software loops for floppy disk I/O. Interrupts occurring during these loops can result in a possible loss of data. Real-time executive routines INTOFF and INTON can be used before and after the floppy disk I/O segments. In our sample application program, executive function is not degraded by temporarily turning interrupts off as long as floppy disk I/O is performed only after data collection has been completed.

The theory of the executive

The executive is composed of five modules: an initializer, a task scheduler, a time delay processor, an interrupt processor, and various support subroutines.

The initializer is contained in subroutine TASKS. This subroutine is the user's only entry to the executive and is called by the user's once-executed main program. The function of this program is to

- place the task subroutine entry points into the correct table within the executive,
- call the user-supplied initialization routine XINIT, and
- call the executive task scheduler.

The task scheduler

- determines the highest-priority task requesting the CPU,
- restores or saves the tasks' data areas (if required), and
- restores the task registers and stack pointer and branches to the task.

The task scheduler runs interruptibly and appears almost as a "zeroth" task. Certain limited sections of the scheduler are nonreentrant and execute with interrupts disabled.

The task scheduler has one entry point that is available only to the executive. If the task scheduler is interrupted, reentry is made from this point; it is not necessary to save registers and restart the task scheduler from the interrupt point.

Several tables are maintained by the task scheduler. The most important is the task status table. This table contains a single byte for each task indicating that task's status: either OFF, BID or RUNNING, TIME-DELAYED, or SUSPENDED. This table is continually scanned by the scheduler for BID or RUNNING tasks.

The remaining tables are used to store information for task start-up. These tables contain

- the restart address of the task,
- the initial entry point of the task,
- the current value of the task stack pointer, and

• the task stack pointer origin.

The processor stack pointer requires special treatment. The executive maintains a separate stack for each task and uses these stacks for saving the task registers whenever an interrupt occurs (more about this later). The executive uses a completely separate stack for its own operations.

The data save areas are maintained within the user's XIOS. The task scheduler contains the routines that shuffle these data areas whenever tasks are changed. When a new task is started, the data areas from the last task executed are saved in that task's corresponding save area in the XIOS. The data stored previously for the new task are then loaded into the correct areas. Two special situations can arise here. First, if the task is unchanged, then no data area swapping will be required. Second, if the task is being executed for the first time (or is being reBID to restart at the initial entry point), then no data swapping will need to be performed, since the XIOS save area will be empty. The first condition is tracked by two single-byte pointers which contain, respectively, the number of the last task executed and the number of the next task. A simple comparison of these two values is used to decide what data swapping will be performed. The second condition is tracked by a data save status table. If the respective task entry in the table is zero, then the task either was just bid for the first time or was reBID. The table entry is changed to a nonzero value just before the task is initiated.

The tables within the task scheduler are initialized by the BID routine. When a task is BID, the following sequence of events occur:

• The status of the BID task is checked. If the BID task is inactive, then the status is changed to active. If the BID task is in any other state, then the BID is ignored.

• The initial task entry point is moved from the entry point table to the restart address table.

• The task stack pointer origin address is moved from the stack pointer origin address table to the stack pointer save table.

• The task data save status table entry is reset to indicate that this task has just been BID.

The time delay processor consists of two routines: one for initiating time delays from a task and one (an internal subroutine) that is used by the executive. Both routines share a common table that contains the time delay value for the task. The value stored here corresponds to the number of clock ticks that the task is to be delayed. A full word is employed to provide for a reasonably large time delay.

Many professionals with limited programming expertise have utilized EXEC-80 for a variety of reasonably complex laboratory applications.

Whenever a clock interrupt occurs, the interrupt processor sets a flag in memory to inform the executive. The executive calls the internal time delay processor whenever this flag is set. This routine checks the status of each task from the task status table to determine which tasks are being time-delayed. For each task that is in time delay, the time delay value is obtained from the table, decremented, and then replaced in the table. If the value is zero or negative, the task status is changed to BID. Only the task scheduler has the authority to select tasks for execution.

A time delay is initiated from a task by the routine TMDL. This routine saves the registers, stack pointer, and restart address in the proper tables and also moves the specified time delay interval into the time delay table. The task status table is changed to indicate that the calling task is now time-delayed. The task scheduler is called at the completion of this routine.

The interrupt processing routine is called by the hardware interrupt system. This routine

• saves the registers, restart address, and stack pointer of the calling task,
• sets a flag indicating to the executive that a clock interrupt has occurred,

- saves the number of the task interrupted in the last task indicator,
- calls the user-supplied routine XSWITCH to determine the present switch positions,
- calls the user-supplied routine XLIGHT with a bit pattern indicative of the task status,
- calls the user-supplied interrupt acknowledge routine, XINT, and
- jumps to the task scheduler.

Most of the coding in the interrupt processor is devoted to a switch-processing algorithm. This algorithm is alerted only when actual switch positions change from the previous value. If a switch has been changed to ON, then the corresponding task is BID or UNSUSPENDED. (A BID starts the task from the initial entry point; an UNSUSPEND simply restarts it from where it was terminated.) Both the BID routine and the UNSUSPEND routine must be called to ensure that tasks are correctly started or restarted. It must be remembered that a task can only be BID if it is inactive. Similiarly, a task can only be UNSUSPENDED if it is SUSPENDED. This prevents any possible conflict. If the switch is now OFF, the task is SUSPENDED.

An important part of the executive is the method used to save the registers, restart address, and stack pointer. Since this procedure is of utmost importance, I will provide complete details for implementing it on an 8080A processor.

When an interrupt occurs, the 8080A processor pushes the program counter on the stack. Thus, the hardware interrupt is functionally equivalent to a software CALL instruction.

The interrupt processing software must take immediate action to ensure that the registers are saved. An appropriate programming response upon entry to the interrupt processor would be

```
XTHL            ; Exchange H and L with top of
                ;   stack. H and L now contain
                ;   task restart address.
PUSH   D        ; Save D and E.
PUSH   B        ; Save B and C.
PUSH   PSW      ; Save A and flags.
PUSH   H        ; Save restart address on stack.
```

All of these registers are saved in the interrupted task's own stack. The XTHL and PUSH H sequence ensures that the task restart address is saved at the top of the stack so that it will be readily available later.

The next programming steps involve saving the restart address and stack pointer in the appropriate tables. The restart address is obtained by a simple POP off the stack. The stack pointer value can be obtained by the following sequence of instructions:

```
LXI    H,0      ; Zero H and L.
DAD    SP       ; Add stack pointer to
                ;   H and L.
                ; H and L now contains stack
                ;   pointer address.
```

The stack pointer and restart address can be saved in a table using code such as the following (this example applies only to the case of the restart address):

```
POP    B        ; Get restart address in B and C.
LDA    TASK     ; Get current task number.
ADD    A        ; Double.
MOV    E,A      ; E = A.
MVI    D,0      ; Zero D.
LXI    H,TABLE-2 ; Get base of table.
DAD    D        ; Add displacement to get actual
                ;   entry address in H and L.
MOV    M,C      ; Move first byte into table.
INX    H        ; Increment pointer.
MOV    M,B      ; Move second byte into table.
```

The software required to restore the registers, stack pointer, and program counter is considerably more complex. The registers themselves must be used during this procedure, complicating the software somewhat. The entire procedure is demonstrated in Figure 7. The restart address is first loaded into register pair D and E and then exchanged with H and L. H and L is subsequently pushed onto the stack. The stack pointer is then loaded into register pair D and E and again exchanged with H and L. The stack pointer is subsequently set by using the SPHL instruction to load it from register pair H and L. The restart address is popped off the stack into D and E and exchanged with H and L. The PSW and registers B and D are then restored and, finally, H and L is exchanged with the top of the stack. All of the registers are now restored, with the restart address found at the top of the register stack. A simple RET instruction completes the sequence.

```
LDA   TASK       ; Get current task number.
ADD   A          ; Double task number.
MOV   E,A        ; Move task number to E.
MVI   D,0        ; Zero second register of pair.
PUSH  D          ; Save displacement.
LXI   H,START-2  ; Get start address table base.
DAD   D          ; Actual address now in H and L.
MOV   E,M        ; Get first byte.
INX   H          ; Increment address pointer.
MOV   D,M        ; Get second byte.
XCHG             ; Get task start address in
                 ;   H and L.
POP   D          ; Restore displacement in table.
PUSH  H          ; Save task start address.
LXI   H,SPSAV-2  ; Get base address of stack
                 ;   save table.
DAD   D          ; Compute actual address.
MOV   E,M        ; Get first byte in E.
INX   H          ; Increment address.
MOV   D,M        ; Get second byte.
XCHG             ; Stack pointer in H and L.
POP   D          ; Task address pointer in D and E.
SPHL             ; Restore stack pointer.
XCHG             ; Get task address in H and L.
POP   PSW        ; Restore PSW.
POP   B          ; Restore B and C.
POP   D          ; Restore D and E.
XTHL             ; Restore H and L, restart address
                 ;   on stack.
EI               ; Enable interrupts.
RET              ; Go to task.
```

Figure 7. Procedure for restoring registers, stack pointer, and program counter.

A number of support programs are provided in the support module of the executive.

Routines SUSPND and UNSPND are used to SUSPEND and UNSUSPEND tasks. Routine SUSPND simply saves the registers, stack pointer, and program counter and then changes the task status in the task status table to SUSPENDED. This routine then returns to the task scheduler. Routine UNSPND is used by a task to UNSUSPEND another task. This routine changes the task status in the task status table to ACTIVE and returns to the calling task.

Routine STOP is used to terminate a task. This routine changes the task status to IDLE and also clears the device status table of any active entries that have been declared by the task to be terminated. This routine returns to the task scheduler.

Routines BSY, NOTBSY, and DVSTAT are used in conjunction with the status declarations discussed earlier. These routines simply use a device status table to track the devices declared by the caller. The device status table contains either a zero, if the device is available, or the number of the task using the device. BSY checks whether the requested device is in use. If the device is not available, then the calling task is time-delayed until it becomes available. If the device is available, then the number of the requesting task is placed in the table and execution is returned to the calling task. Routine NOTBSY zeroes the device status table entry, regardless of its present status. Routine DVSTAT is used to check on the status of a device. It returns either a zero if the device is not in use or the number of the task using the device if it is in use.

Routine TSKNO returns the task number of the calling program. This routine can be useful for displaying task information or for developing routines that are task-number-independent.

Routines INTON and INTOFF are useful for controlling interrupts within the system. Some floppy disk controllers rely on software-timed loops during read or write operations. Any interrupts occurring within these loops lead to timing difficulties. Routines INTON and INTOFF can be used to ensure that interrupts do not occur during floppy disk reads or writes.

The unique contribution of the real-time executive presented here is that it is designed to interface to an existing, off-the-shelf Fortran compiler and operating system. Furthermore, its routines are modularized to provide maximum transportability and flexibility with minimum assembler language interfacing required from the user.

Several years of experience with the EXEC-80 system have demonstrated its abilities. Many professionals with limited computer programming expertise have utilized it for a variety of reasonably complex laboratory applications.

Availability

The EXEC-80 system is available in object-relocatable form from Available Energy, Inc., 5724 Cass Ave., Detroit, MI 48202. There is a $50 preparation charge. The system is designed to interface to CP/M 2.2 and Microsoft Fortran 3.44 (not provided). It is available only on 8-inch, single-density, CP/M-compatible disks.

Acknowledgment

The author would like to acknowledge Available Energy, Inc., of Detroit for its support during the development of EXEC-80.

Daniel A. Crowl is associate professor of chemical engineering at Wayne State University in Detroit and also director of the College of Engineering Computer Graphics and Design Laboratory. His interest in computers has developed as a result of extensive work in chemical engineering simulation and process control implementation.

He obtained a BS in fuel science from Pennsylvania State University in 1971 and an MS and PhD in chemical engineering from the University of Illinois in 1973 and 1975, respectively. He is a member of the American Institute of Chemical Engineers, the American Chemical Society, and Sigma Xi.

Questions about this article can be directed to Crowl at the Department of Chemical Engineering, Wayne State University, Detroit, MI 48202.

References

1. D. A. Mellichamp, "An Introduction to Real-Time Computing," *Cache Monograph Series in Real-Time Computing,* D. A. Mellichamp, ed., Cache, Cambridge, MA, 1979.
2. W. Crutchley, "A Real-Time Executive for Your Microcomputer," *Programming Techniques, Vol. 4: Bits and Pieces,* B. L. Liffick, ed., Byte Books, McGraw-Hill, New York, 1979.
3. "Industrial Computer System Fortran Procedures for Executive Functions and Process Input-Output" (Instrument Society of America Standard S61.1), ISA, Pittsburgh, PA, 1972.

Mapping High-Level Syntax and Structure Into Assembly Language

M. F. Smith
Istel Limited

Y. Hoffner
University of Reading

M. A. Sealey
Plessey Radar

Does the mapping of mnemonics into high-level notation make assemblers easier to use? The authors' experience suggests that it does not.

It has been long accepted that mnemonic assembler programs are difficult to write and read.[1] Assemblers provide few aids for delineating the structure of programs and data, and so the overall organization of assembler software is usually obscure. Programming errors can be easily made and, in most assemblers, error detection mechanisms are primitive; this contributes to low productivity and to difficulty in testing and maintaining programs.

Learning an assembler is made more difficult by mnemonics which may be obscure and difficult to remember. Practically every processor has its own set of mnemonics, in some cases radically different from the others, and thus there can be little portability of programs or of programmers.[2] For these reasons, and probably others, high-level languages have for the most part replaced assemblers in computer applications in which execution speed is not critical.[1]

Microprocessors have revived assembly language, since many of them have lacked high-level-language support for some time after their introduction.[3] This has had the effect of creating a body of programmers who often have had no support or inclination for high-level languages; consequently, assemblers have remained important in their programming environments.[4] In some cases, the use of assemblers is justified easily; this particularly is the case in dedicated (embedded) microprocessor applications—especially those employing single-chip microcomputers.[3] This may be attributed to several problems associated with using high-level languages in dedicated microprocessor applications[5]:

- Many languages cannot produce programs which can be placed in ROM; ROM storage usually is necessary in dedicated applications.
- The size of run-time programs may be too large for the memory available; single-chip microcomputers have limited memory, and expansion off-chip may be uneconomical.
- Interrupt and restart vectors may not be accessible; dedicated applications typically are in real time, use interrupts, and require power-on start-up.

0272-1732/85/0800-0067$01.00 © 1985 IEEE

• Bit and address manipulations may be difficult to program in high-level languages; dedicated applications usually require these operations because of their use of highly specialized I/O devices.
• High-level languages may presuppose the existence of a sophisticated operating system; dedicated systems tend not to have separate operating systems.
• High-level language programs can be extremely difficult to debug at machine level; dedicated microcomputers almost always require debugging at this level because of their use of specialized hardware for I/O.

It should be stressed, too, that assembly language may be more suitable, from a conceptual viewpoint, for solving some problems than many high-level languages.[1,6] When used appropriately, assemblers can be more productive and readable than high-level languages used inappropriately.[4,7] It can be expected, therefore, that assembly language, either embedded within high-level-language programs or standing by itself, will continue to be used extensively in certain applications. Consequently, it is probably worthwhile to improve the syntax and structure of assemblers so that they can be used more effectively in their proper context.

Sometimes assembly language can actually be more productive and readable than high-level language.

Improving assembly language

Before we discuss improving assembly language, we should digress slightly and examine what constitutes desirable features in any language. Attributes which have been judged important are

• a syntax which is clearly defined, logical, regular, and unambiguous,[8,9,10]
• readability,[10]
• operations which are appropriate for solving the problem,[9]
• reduction of potential syntactical and structural errors,[11]
• data-structuring facilities,[8]
• writeability, i.e., ease of program creation,[10] and
• handling of minor programming details not pertinent to the solution of the problem.[9]

It should come as little surprise that classical mnemonic assemblers exhibit few of these features. In many cases, even a logical and regular syntax is absent.[2,4]

Efforts to improve assemblers have been directed largely toward producing a universal syntax. Some efforts have concentrated on defining standard mnemonics,[2,6,12] while others have replaced mnemonics with symbolic or algorithmic notations based on high-level-language syntax.[4,7,13-15] Thus far, the improvement of assemblers seems to have involved

• removing inconsistencies within assembler sets,[2,4]
• simplifying benchmarking and selection of processors,[12]
• eliminating conflicting usage between assemblers,[2,4]
• promoting portability between processors,[2,4,12,13]
• improving readability of assemblers,[2,4]
• making assemblers easier to learn,[2,4,12] and
• improving the productivity of programming in assembler.[2]

It must be recognized that efforts at improving assemblers[2,4,12,13] have resulted in far more regular and consistent instruction sets. It is arguable whether the other aspects have actually been improved or not.

Improving assembler with high-level-language syntax

The use of high-level-language syntax in place of mnemonics is surprisingly uncommon, in spite of the fact that manufacturers frequently explain their mnemonics with symbolic

notation.[15] SMAL/80[15,16] achieves an elegant symbolic representation of an 8080 assembler, but strays into the realm of high-level language by providing programmed control structures. A-Natural[14] does much the same since some of its instructions translate to more than one machine code instruction; its syntax appears rather opaque, as well.

The work of Duncan[4,7] in defining a machine-independent syntax suitable for the 8080, Z80, and MC6800 is comprehensive and appears to be implementation-oriented. The syntax, however, does not seem sufficiently similar to any high-level language to be immediately recognizable as such. There seem to be syntactical irregularities and special cases; for example, A:=0, but A:=A+1 is represented as A:+1. Turner[13] points to addressing problems in Duncan's notation. Many of the instructions of Duncan's symbolic notation[4,7] appear even more opaque than the instructions they replace; for example, the MC6800 instruction ROLA is expressed as AK:@L in Duncan's notation.[4] The use of labels seems more primitive than that of most mnemonic assemblers in that only numerical labels are permitted and, apparently, only numerical offsets may be used for branch statements for the MC6800.[4] Moreover, Duncan's notation does not appear highly readable (Figure 1).

PASSEM[13] successfully maps the mnemonics of the 8080/8085 and MC6800 microprocessors into a high-level-language syntax based on Pascal.[8] There still remain syntactical irregularities; for example, the MC6800 instruction CLRA is represented as (A):=ZERO, not (A):=0. The use of parentheses to indicate the contents of registers seems unnecessary and tedious since there is no other possible interpretation of the use of a register. The overall readability of PASSEM (see Figure 1 again) does appear superior to that of Duncan's notation, however.

Implementing an assembler with a high-level-language syntax

After examining previous attempts at incorporating high-level-language syntax into assembly language, we decided that it might be worth attempting a similar type of assembler for the Motorola MC6809 microprocessor.[11] Our assembler, called IASM, was programmed in Pascal on a PDP-11/44 under UNIX Version 7. We implemented IASM as a translator which produces mnemonic assembler output for a conventional cross-assembler. We did this because we intended IASM to serve more as a test of the concepts of symbolic assemblers than as a production software tool.

In attempting to map an assembler syntax into a high-level-language syntax, we selected Pascal and related languages such as PLZ/SYS[9] for our models. It is possible that Basic or Fortran, which are closer to assembler in many respects and better known, could have made the transition simpler.[17] However, they lack many features which contribute to good programming techniques, e.g., long variable names, named labels for GOTOs, enforced variable declaration, and data structures.

Classical mnemonic assemblers exhibit few of the attributes judged to be important in computer languages.

Almost all statements of an assembler can be mapped into high-level syntax in one fashion or another (see appendix). But there is a point, soon reached, at which this syntax can become cryptic, tedious, or misleading. For example, PUSH and PULL can be treated as STORE and LOAD with autodecrement; in the MC6809, PSHS A,X can be represented quite logically as S(---):=A,X. However, we chose the notation which seemed more obvious—we used a mnemonic. High-level notation can also be misleading; for example, A:=A*2 (LSLA) can lead an inexperienced programmer to believe that this is a general-purpose multiplication instruction rather than a shift. The implementation of IASM is not free of irregularities, either. An assembler statement such as DECA was implemented as A=A−1 (see appendix), which is inconsistent with instructions of the type A=A+ IMMED(val). It should have been implemented as A=A−IMMED(val) so the improved assembler could recognize the special case of A=A−IMMED(1) and produce a

MC6802 assembler program			Duncan's program		PASSEM program
RESULT	RMB	1	NULL;		RESULT: RESERVE 1 BYTE;
DEST	RMB	2	NULL;		DEST: RESERVE 2 BYTES;
TABLE	RMB	2	NULL;		TABLE: RESERVE 2 BYTES;
BINDEC	PSHA		M[SP]: = A;SP: − 1;		((SP)): = (A);
	PSHB		M[SP]: = B;SP: − 1;		((SP)): = (B);
	STX	DEST	MM0001: = IX;		(DEST): = (X);
	LDX	#TENS	IX: = 013A;		(X): = TENS;
NEXT	CLR	RESULT	M0000: = 0;	NEXT:	(RESULT): = 0;
SUB16	SUB B	1,X	B: − M1X;	SUB16:	(B): = (B) − (1 + (X));
	SBC A	0,X	A: − − M0X;		(A): = (A) − (0 + (X)) − (CY);
	BCS	ADD16	JK + 05; (to ADD16)		IF CY GOTO ADD16;
	INC	RESULT	M0000: + 1;		(RESULT): = (RESULT) PLUS 1;
	BRA	SUB16	J + F5; (to SUB16)		GOTO SUB16;
ADD16	ADD B	1,X	B: + M1X;	ADD16:	(B): = (B) + (1 + (X));
	ADC A	0,X	A: = + + M0X;		(A): = (A) + (0 + (X)) + (CY);
	PSHA		M[SP]: = A;SP: − 1;		((SP)): = A);
	STX	TABLE	MM0003: = IX;		(TABLE): = (X);
	LDX	DEST	IX: = MM0001;		(X): = (DEST);
	LDA A	RESULT	A: = M0000;		(A): = (RESULT);
	ADD A	#30H	A: + 48;		(A): = (A) + HEX 30;
	STA A	0,X	M0X: = A;		((X)): = (A);
	INX		IX: + 1;		(X): = (X) PLUS 1;
	STX	DEST	MM0001: = IX;		(DEST): = (X);
	PULA		SP: + 1;A: = M[SP];		(A): = ((SP));
	LDX	TABLE	IX: = MM0003;		(X): = (TABLE);
	INX		IX: + 1;		(X): = (X) PLUS 1;
	INX		IX: + 1;		(X): = (X) PLUS 1;
	CPX	#TENSEND	IX − 0144;		TEST (X) − TENSEND;
	BNE	NEXT	JNZ + D1; (to NEXT)		IF NOT ZERO GOTO NEXT;
	PULB		SP: + 1;B: = M[SP];		(B): = ((SP));
	PULA		SP: + 1;A: = M[SP];		(A): = ((SP));
	RTS		RET;		
TENS	FDB	10000	#2710#	TENS:	STORE DEC DATA AS 2 BYTES
	FDB	1000	#03E8#		10000,1000,100,10,1;
	FDB	100	#0064#		
	FDB	10	#000A#		
	FDB	1	#0001#		
TENSEND	FDB	255	#00FF#	TENSEND:	STORE DEC DATA AS 2 BYTES

Figure 1. A binary-to-decimal conversion routine coded in Duncan's high-level-language syntax assembler,[6] Turner's PASSEM high-level-language syntax assembler,[12] and conventional MC6802 mnemonic assembler.

DECA for it. In the case of A = A − IMMED(55H) it should produce SUBA #55H. Similar criticism can be leveled against an instruction such as A:COMP: (see appendix), which should have been implemented as A = :COMP:A or A = COMP(A). The reason for such syntactical irregularities was a compromise made for the sake of an easier implementation.

Mapping an assembler into high-level syntax can provide the opportunity to make instruction sets appear more orthogonal and therefore simpler. For example, other authors[13-15] have distinguished syntactically between instructions such as INCREMENT and CLEAR and their respective arithmetic and assignment operations. In the syntax of IASM, these are not treated as special instructions but as more efficient special cases. This does have the potential of leading to errors if programmers forget that while COUNT: = COUNT + 1 is legal, COUNT: = COUNT + 2 is not. To handle the latter automatically is tempting, but this would be stepping into the realm of high-level languages.

Confusion of immediate operands and addresses with variables is an error frequently encountered in assemblers[12,13] (Figure 2). Such errors are especially dangerous because they are very difficult to detect and can escape even the scrutinizing eyes of an experienced programmer. This danger may be reduced by improving the syntax of addressing modes.[2,13] In IASM, immediate operands are prefixed with the operator IMMED, e.g., X = IMMED(25), and memory references are prefixed by the operator MEM, e.g., X = MEM(25). This may seem heavy-handed, but it is almost certainly unequivocal. In this case, the improvement of the syntax of assemblers can be seen as a genuine aid to reducing errors in programming, especially errors which can be difficult to detect (Figure 2).

Intended Code

```
                Object
Location        code line          Source line
                                1 "6809"
                        (0037)  2 VAL     EQU     55
0000                            3 CHAR    RMB     1
0001                            4 TABLE   RMB     2
                                5
0003 8E 0000                    6         LDX     #CHAR
                                7
0006 8E 0001                    8         LDX     #TABLE
0009 E6 84                      9 LOOP:   LDB     0,X
                               10 ;       .
                               11 ;       .
000B 86 37                     12         LDA     #VAL
                               13 ;       .
                               14 ;       .
000D 83 1234                   15         SUBD    #1234H
Errors =  0
```

Code produced by conventional assembler (with errors)

```
                Object
Location        code line          Source line
                                1 "6809"
                        (0037)  2 VAL     EQU     55
0000                            3 CHAR    RMB     1
0001                            4 TABLE   RMB     2
                                5
0003 BE 0000                    6         LDX     CHAR
                                7
0006 BE 0001                    8         LDX     TABLE
0009 E6 84                      9 LOOP:   LDB     0,X
                               10 ;       .
                               11 ;       .
000B 96 37                     12         LDA     VAL
                               13 ;       .
                               14 ;       .
000D B3 1234                   15         SUBD    1234H
Errors =  0
```

Figure 2. Intended code (top) and the code produced by a conventional assembler (bottom). The errors shown commonly occur when the instructions accessing memory are not uniquely defined; they can be the result of a mistake in defining an immediate operation.

Evaluation of assemblers with high-level syntax

To evaluate the readability of symbolic assemblers, we programmed a well-known

binary-to-decimal routine[11] in IASM (Figure 3), PASSEM (Figure 1), and Duncan's notation (Figure 1). Duncan's notation appears the least readable, but IASM seems only arguably better than PASSEM. What is more interesting about the comparison is that none of the symbolic notations show readability that is outstandingly superior to the MC6809 mnemonic assembler routine (see Figure 3 again). In fact, one of us commented that the assembler included in the comparisons (see Figures 1, 3, and 7) seemed essential for documenting the symbolic programs. This suggests strongly that a well-designed mnemonic assembler may not be inferior to high-level notation.

In high-level languages, explicit type definition and checking prevents many programming errors. Type errors are common in assemblers, although they are not always similar to those found in high-level-language programming. In traditional assemblers, the directives EQU, FCB, FDB, and RMB serve much the same function as the CONST and VAR declarations in Pascal (Figure 4); assemblers simply do not utilize this information to check for type errors or for addressing information. In IASM, variables are declared in a manner

MC6809 assembler program			Equivalent IASM program	
BINDEC:	PSHU	D,X,Y	BINDEC:	PUSH USTACK D,X,Y;
	LDY	#TENS		Y = IMMED(TENS);
NEXT	CLR	,X	NEXT:	MEM(0+X) = 0;
SUB16	SUBD	,Y	SUB16:	D = D − MEM(0+Y);
	BCS	SUB16		IF C = 1 THEN GOTO SUB16;
	ADDD	,Y++		D = D + MEM(Y++);
	PSHU	A		PUSH USTACK A;
	LDA	,X		A = MEM(0+X);
	ADDA	#30H		A = A + IMMED(HEX30);
	STA	,X+		MEM(X+) = A;
	PULU	A		PULL USTACK A;
	TST	,Y		TEST(0+Y);
	BPL	NEXT		IF POS THEN GOTO NEXT;
	PULU	D,X,Y		PULL USTACK A,B,X,Y;
	RTS			RETURN FROM SUB;
TENS:	FDB	10000		STORE TENS: WORD,
	FDB	1000		10000,1000,100,10,1,255;
	FDB	100		
	FDB	10		
	FDB	1		
	FDB	255		

Figure 3. A binary-to-decimal conversion routine coded in conventional MC6809 mnemonic assembler and in IASM high-level-language syntax assembler.

Improved assembler		Conventional assembler			Address
Const	number = 7	;NUMBER	EQU	7	
Var	mice:byte	;M1CE	RMB	1	0
	moles:word	;MOLES	RMB	2	1
	name:string[10]	;NAME	RMB	10	3
Store	table:word,10,20,0	;TABLE	FDB	10,20,0	13
	nobble:byte,100	;NOBBLE	FCB	100	19
	message:string,"Hello"	;MESSAGE	ASC11	"Hello"	20

Legal use of variables and constants in an improved assembler

lda	number	;LDA	#NUMBER	Typing selects
inc	mice	;1NC	M1CE	correct mode
sta	name[number]	;STA	NAME + 7	of addressing.
ldb	nobble	;LDB	NOBBLE	
ldd	number	;LDD	#NUMBER	Typing selects
std	moles	;STD	MOLES	correct mode
addd	table[3]	;ADDD	TABLE + 4	of addressing.
ldx	name	;LDX	#NAME	Index registers
leay	number,u	;LEAY	7,U	normally use
ldu	message	;LDU	#MESSAGE	addresses.
stx	moles	;STX	MOLES	

Deliberate type violations in an improved assembler

ldx	/name	;LDX	NAME	Load X from a
		;		string location.
lda	#mice	;LDA	#MICE	Load A with 0.
lda	/moles	;LDA	MOLES	Load A with MSB
		;		of word variable.
ldd	/name[1]	;LDD	NAME + 1	Load two bytes
		;		of a string.
stx	/name	;STX	NAME	Redundant.

Typical type errors in a conventional MC6809 assembler

LDA	NUMBER	;load from address 7?
STA	MOLES	;storing byte in word location?
LDB	#NOBBLE	;load the word 19 into a byte?
LDD	number	;load from address 7?
STD	M1CE	;store a word in a byte?
ADDD	#MOLES	;add 1 to D?
LDY	TABLE	;probably an error
LDX	M1CE	;load X from a byte variable?
LEAY	NOBBLE,U	;LEAY 19,U?

Figure 4. Suggested methods of definition and manipulation of variables and constants in an improved mnemonic assembler (variable type errors and deliberate type violations are shown).

Static arrays in memory for an improved assembler

```
Type dogs    = array [10] of words     ;DOGS RMB 20
                 .
                 .
     addd      dogs[2]                 ;ADDD DOGS+4
     std       dogs[0]                 ;STD DOGS+0
```

Implicit dynamic arrays effected by index/stack registers

```
Type dogs    = array [10] of words     ;DOGS RMB 20
Type cats    = array [50] of bytes     ;CATS RMB 50
                 .
                 .
     ldx       cats    ;load the Bth element of the array
     lda       X[B]    ;pointed to by X into A, i.e., LDA B,X
                 .
                 .
     ldu       dogs    ;store the word in Y into the sixth
     sty       U[5]    ;element of the array pointed to by U,
                 .     ;i.e., STY 5,U
                 .
     lda       S[-40]  ;load A from the -40th byte of the
                       ;stack, i.e., LDA -40,S
```

Figure 5. Possible methods of definition and manipulation of static and dynamic arrays in an improved mnemonic assembler.

similar to Pascal, but type checking was never implemented (see Figure 4 again); we feel that there is still a good deal of work needed in this area. It would be beneficial to include constructs which could encourage error-free programming of I/O. Such constructs could

Any universal assembler offers, at best, only a limited sort of portability between machines.

define input and output channels separately so that no erroneous writing to an input channel, or reading from an output channel, could take place. The status and control registers usually associated with input and output channels could be protected in a similar way. Such facilities could apply to different processor families regardless of whether input/output devices are defined as memory locations or as special I/O channels.

Arrays and records frequently are employed in assembler programming, but the organization of the data may be concealed because mechanisms for defining complex data structures do not exist. A provision for explicitly defining data structures can be an important aid to understanding program structure in addition to other documentation. Definition of data structures, in a manner similar to Pascal, can be implemented in an assembler without moving into the realm of high-level languages (Figures 5 and 6). The use of static arrays can be simplified by defining them explicitly as arrays rather than by reserving space for them, as is done in conventional assemblers. Defining the array as a list of items of a certain type and length, such as bytes, words, or double words, can aid in referring to the arrays at

a later stage (see Figure 5 again). The programmer does not have to calculate the offset by taking into account the length of the item of which the array consists; the improved assembler can do it for the programmer automatically. An explicit record structure can aid the programmer in understanding the complex manipulation of data structures (see Figure 6 again). It is possible to mimic such structures with conventional assembler code, but such practice, although highly recommended, is tedious, error-prone, and insufficiently explicit. This is something which was planned for IASM but not actually implemented.

The value of high-level syntax in assembler

A universal assembler syntax, either mnemonic or symbolic, should be an improvement from the programmer's viewpoint. It makes it far easier to change from one processor to another, since the bulk of instructions used in assembler programming are common to all machines.[12,18,19] It is highly likely that students[19] or programmers with little experience of assemblers will find high-level-language notation easier to read than mnemonics.[13] It does not necessarily follow, however, that being able to read assembler programs easily will make programming significantly more productive or program maintenance easier. Our reaction, qualitative as it might be, is somewhat to the contrary.

Problems which are solved best by assembler programming are quite simply difficult to solve, no matter what form of language is employed.

Any universal assembler offers, at best, only a limited sort of portability between machines because differences in computer architecture cannot be concealed from the programmer (nor does it seem desirable that they should). For example, the registers of the MC6800 and the Z80, and the manner in which they are

Improved assembler record structure	Conventional assembler: Typical practice		Conventional assembler: Good practice		
Type pets =					
record					
feet:byte;			FEET	EQU	0
paws:word;			PAWS	EQU	1
tail:byte;			TAIL	EQU	3
length:word;			LENGTH	EQU	4
weight:byte;			WEIGHT	EQU	6
bites:byte;			BITES	EQU	7
end record;	PETS:	RMB 8	PETS:	RMB	9
.		.		.	
.		.		.	
.		.		.	
lda pets.weight		LDA PETS+6		LDA	PETS+WEIGHT
ldb pets.bites		LDB PETS+7		LDB	PETS+BITES
mul		MUL		MUL	
std pets.paws		STD PETS+1		STD	PETS+PAWS

Figure 6. Possible means of definition and manipulation of record structures in an improved mnemonic assembler.

MC6800 routine			Z80 routine		MC6809 routine	
RESULT:	RESB					
DEST:	RESH					
TABLE:	RESH					
BINDEC:	ST	X,/DEST				
	LD	X,#TENS	LD	IY,#TENS	LD	Y,#TENS
NEXT:	CLR	/RESULT	XOR	A	CLR	0(X)
SUB16:			SETC			
			NOTC			
	SUB	B,1(X)	LD	D,1(IY)	SUB	D,0(Y)
	SUBC	A,0(X)	LD	E,0(1Y)		
			SUBC	HL,DE		
	BC	ADD16	BC	ADD16	BC	ADD16
	INC	/RESULT	INC	A	INC	0(X)
	BR	SUB16	BR	SUB16	BR	SUB16
ADD16:	ADD	B,1(X)	ADD	HL,DE	ADD	D,++(Y)
	ADDC	A,0(X)				
	PUSH	A			PUSHS	A
	ST	X,/TABLE				
	LD	X,/DEST				
	LD	A,/RESULT			LD	A,0(X)
	ADD	A,#30H	ADD	A,#30H	ADD	A,#30H
	STA	A,0(X)	ST	A,0(1X)	ST	A,(X)+
	INC	X	1NC	IX		
	ST	DEST,/X				
	POP	A			POPS	A
	LD	X,/TABLE				
	INC	X	1NC	IY		
	INC	X	INC	IY		
	CP	X,#TENSEND	CP	IY,#TENSEND	CP	Y,#TENSEND
	BNZ	NEXT	BNZ	NEXT	BNZ	NEXT
	RET		RET		RET	
TENS:	DATAH	10000	DATAH	10000	DATAH	10000
	DATAH	1000	DATAH	1000	DATAH	1000
	DATAH	100	DATAH	100	DATAH	100
	DATAH	10	DATAH	10	DATAH	10
	DATAH	1	DATAH	1	DATAH	1
TENSEND:						

Figure 7. Comparison of the same program, BINDEC,[10] coded in the proposed IEEE standard mnemonic assembler[2] for the MC6800, the Z80, and the MC6809. The routines have been converted manually into the standard assembler.

used, are so different that portability between the two appears highly unlikely (Figure 7). A real danger of apparent portability at this level of programming is that less obvious differences between processors, such as flag treatment (Figure 8), can introduce subtle flaws when programs are transported.[20]

Symbolic assemblers seem better for representing mathematical and data-movement operations,[19] while mnemonics seem more appropriate for representing the remaining types of instructions. In fact, the symbolic assemblers of Duncan and Turner, and ours, generally revert to mnemonic notation (or opacity) once arithmetic and data-movement instructions have been defined. If the statistical averages of machine-instruction usage derived by Fairclough[18] are taken into account, it may be supposed that 55 percent of all instructions can be better represented by symbolic notation, while the remaining 45 percent can be more suitably represented in a mnemonic form.

That assemblers have remained virtually unchanged for years[4] is empirical evidence in favor of mnemonics. It is also possible that problems which are solved best by assembler language may be unsuited to mathematical expression. The case for symbolic notation appears marginal; therefore, we feel that efforts to improve the syntax of assemblers may as well remain within the traditional framework of mnemonics and that efforts to improve assembler language should be directed elsewhere. This is not to depreciate efforts at improving syntax, since there is still much to be done in this area. It simply appears to us that mnemonics are more practical for assembler programming than symbolic high-level notation.

Possibly the gravest objection to assembler programming is its lack of constraints on program structure. The lack of consistent program structure and of program modularity results in software which is difficult to test, maintain, and extend. Common weaknesses in assembler programs are uncontrolled entries to, and exits from, subroutines. Krieger[15] has claimed that the employment of high-level syntax in an assembler automatically confers the benefits of structured programming and modularity. This is misleading, since it is possible to employ these techniques in any language, even a mnemonic assembler. Structured programming is a method and, as such, is not affected as much by language design as by intellectual discipline and good planning. It is true, however, that the provision of certain software mechanisms[21] can aid, or even enforce, program structure. Unfortunately, these mechanisms do not appear to have been incorporated directly into any popular assemblers to date. It is likely that they would be advantageous, but the methods are still undefined; further study is planned.

Our experience with IASM indicates that symbolic notation enjoys little advantage over conventional mnemonic assemblers. We would suggest, however, that facilities such as data structures and the typing of variables, incor-

Z80	Flags SZHVC	MC6809	Flags nzhvc	IEEE proposed standard mnemonic
ADC	↑ ↑ ↑ ↑ ↑	adc	↑ ↑ ↑ ↑ ↑	ADDC
ADD	↑ ↑ ↑ ↑ ↑	add	↑ ↑ ↑ ↑ ↑	ADD
AND	↑ ↑ 1 ↑ 0	and	↑ ↑ * 0 *	AND
CALL	* * * * *	jsr	* * * * *	CALL
CP	↑ ↑ ↑ ↑ ↑	cmp	↑ ↑ ? ↑ ↑	CMP
DEC	↑ ↑ ↑ ↑ *	dec	↑ ↑ * ↑ *	DEC
INC	↑ ↑ ↑ ↑ *	inc	↑ ↑ * ↑ *	INC
JP	* * * * *	jmp	* * * * *	JMP
LD	* * * * *	ld	↑ ↑ * 0 *	LD
		st	↑ ↑ * 0 *	ST
POP	* * * * *	pul	* * * * *	POP
PUSH	* * * * *	psh	* * * * *	PUSH
RET	* * * * *	rts	* * * * *	RET
RRA	* * 0 * ↑	rora	↑ ↑ * * ↑	ROR
SBC	↑ ↑ ↑ ↑ ↑	sbc	↑ ↑ ? ↑ ↑	SUBC
SUB	↑ ↑ ↑ ↑ ↑	sub	↑ ↑ ? ↑ ↑	SUB

Figure 8. Some common instructions for the Z80 and the MC6809, and their corresponding flag operations; the proposed IEEE standard mnemonics are shown as well. Symbols indicating flag operations are * = unchanged, ↑ = affected appropriately, 0 = cleared, and ? = undefined.

porated into the framework of conventional mnemonic assemblers, appear to be worthwhile. Such facilities would affect assembler directives and the internal mechanisms of assemblers rather than the design of the actual instruction sets. Given this, it should be possible to incorporate typing and data structures with a minimum of upset to programmers who are accustomed to traditional mnemonic assemblers. It seems likely that the flagging of potential addressing errors would soon gain the confidence of users; whether or not explicit record structures would ever be used is less certain. In addition, typing of variables would allow instruction sets to be simplified, since there would be no need to distinguish between immediate operands, bytes, or words. This could be important when dealing with newer microprocessors which support bits, BCD digits, bytes, words, double words, and quad words.

It is well to bear in mind that most of the difficulty of using assemblers lies as much in the nature of the applications which are suitable for them as in their shortcomings. Problems which are solved best by assembler programming are quite simply difficult to solve, no matter what form of language is employed. There is, however, a great need for improvement of assembly language syntax, though such improvement should remain within the traditional framework of mnemonic assemblers. There is also a strong case for changing the structure of assemblers by including typing of variables and data structures. Such improvements, both to the syntax and structure of assemblers, will be useful, but they are not likely to overcome the inherent challenge of assembler programming.

Acknowledgment

The authors would like to thank J. D. Roberts and R. M. Yorston of the Department

Appendix—Partial and informal definition of IASM

Operation	6809 mnemonic	IASM instruction
Addition	ADDA #1AH	A = A + IMMED(HEX 1A);
	ADDA Var	A = A + MEM(Var);
	ADDA #Var	A = A + IMMED(Var);
	ADCA 5,X	A = A + MEM(5 + X) + C;
	ADCA [5,X]	A = A + [MEM(5 + X)] + C;
	ADDD Var	D = D + MEM(Var);
Decimal adjust	DAA	DECIMAL ADJUST;
Decrement	DECA	A = A − 1;
	DEC Var	MEM(Var) = MEM(Var) − 1;
	DEC [Var]	[MEM(Var)] = [MEM(Var)] − 1;
And	ANDA #55	A = A:AND:IMMED(55);
Or	ORA Var	A = A:OR:MEM(Var);
Complement	COMA	A:COMP:;
	COM Var	MEM(Var):COMP:;
Bit test	BITA Var	STATUS(A:AND:MEM(Var));
Test	TSTA	STATUS(A − IMMED(0));
Compare	CMPA #0AH	STATUS(A − IMMED(HEX A));
	CMPY Var	STATUS(Y − MEM(Var));

of Computer Science of the University of Reading, and J. B. V. Smith of Reading, for their helpful comments on the subject and for their editing of the manuscript.

References

1. J. E. Sammet, *Programming Languages: History and Fundamentals,* Prentice-Hall, Englewood Cliffs, NJ, 1969.
2. W. P. Fischer, "Microprocessor Assembly Language Draft Standard," *Computer,* Vol. 12, No. 12, Dec. 1979, pp. 96-109.
3. P. Caudill, "Using Assembly Coding to Optimize High-Level Language Programs," *Electronics,* Vol. 53, No. 3, Feb. 1, 1979, pp. 121-124.
4. F. G. Duncan, "Level-independent Notation for Microcomputer Programs," *IEEE Micro*, Vol. 1, No. 2, May 1981, pp. 47-52.
5. R. E. James III, "A Proposed Standard for Extending High-Level Languages for Microprocessors—IEEE Task P755," *IEEE Micro,* Vol. 1, No. 2, May 1981, pp. 70-75.
6. M. Biewer, "The STD Instruction Mnemonics: A Proposal for Standard Mnemonics," in *Microprocessor User's Guide,* Pro-Log Corp., 1979, pp. 55-63.
7. F. G. Duncan, *Microprocessor Programming and Software Development,* Prentice-Hall International, London, 1979.
8. K. Jensen and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag, New York, 1975.
9. T. Snook et al., *Report on the Programming Language PLZ/SYS,* Springer-Verlag, New York, 1978.
10. T. W. Pratt, *Programming Languages: Design and Implementation,* Prentice-Hall, Englewood Cliffs, NJ, 1975.
11. M. F. Smith, "Comparative Software Analysis of the MC6809 Microprocessor," *Microprocessors and Microsystems,* Vol. 5, No. 9, Nov. 1981, pp. 401-404.
12. J. D. Nicoud, "A Common Microprocessor Assembly Language," *Proc. Second Euromicro Conf.,* Venice, Oct. 1976, pp. 1-7.

Operation	6809 mnemonic	IASM instruction
Clear	CLRA	A = 0;
	CLR Var	MEM(Var) = 0;
	CLR [Var]	[MEM(Var)] = 0;
Load	LDA Var	A = MEM(Var);
	LDA #Var	A = IMMED(Var);
	LDU D,Y	U = MED(D + Y);
	LDB ,X++	B = MEM(X++);
	LDB [,X++]	B = [MEM(X++)];
	LDX [55,Y]	X = [MEM(55 + Y)];
	LDU [D,Y]	U = [MEM(D + Y)];
	LDS 0,X	S = MEM(0 + X);
Load effective address	LEAY 5,X	Y = 5 + X;
	LEAY D,X	Y = D + X;
Store	STY Var	MEM(Var) = Y;
	STA −5,U	MEM(−5 + U) = A;
	STA [−5,U]	[MEM(−5 + U)] = A;
Multiply	MUL	D = A*B;
No operation	NOP	NO OPERATION;

13. G. Turner, "PASSEM—A Universal Approach to Assembly Language," *Microprocessors and Microsystems,* Vol. 5, No. 9, Nov. 1981, pp. 395-399.

14. M. S. Krieger and P. J. Plauger, "C Language's Grip on Hardware Makes Sense for Small Computers," *Electronics,* Vol. 53, No. 11, 1980, pp. 129-133.

15. M. S. Krieger, "Structured Assembly Language Suits Programmers and Microprocessors," *Electronics,* Vol. 53, No. 1, Jan. 17, 1980, pp. 118-122.

16. A. Mosak, "Structured Programming *Can* Be Applied to Microprocessors—Even by Novices: A Review of *Structured Microprocessor Programming*," *IEEE Micro,* Vol. 2, No. 1, Feb. 1982, pp. 63-71.

17. W. P. Fischer, "Forget Mnemonics?—Chairman's Reply" (letter), *Computer,* Vol. 14, No. 5, May 1981, p. 9.

18. D. A. Fairclough, "A Unique Microprocessor Instruction Set," *IEEE Micro,* Vol. 2, No. 2, May 1982, pp. 8-18.

19. F. G. Duncan, "Forget Mnemonics?" (letter), *Computer,* Vol. 14, No. 5, May 1981, p. 8.

20. M. F. Smith and B. E. Luff, "Automatic Assembler Source Translation from the Z80 to the MC6809," *IEEE Micro,* Vol. 4, No. 2, Apr. 1984, pp. 3-9.

21. S. Kawai, "A Semiblock Structure for Low-Level Languages," *Software—Practice & Experience,* Vol. 10, 1980, pp. 11-19.

Operation	6809 mnemonic	IASM instruction
Shift right	LSRA	A = A/2;
	LSR Var	MEM(Var) = MEM(Var)/2;
	LSR [Var]	[MEM(Var)] = [MEM(Var)]/2;
Rotate right	RORA	A = A/2/C;
Rotate left	ROLA	A = A*2*C;
Push registers onto stack	PSHS A,B,X	PUSH SSTACK A,B,X;
Pull registers from stack	PULU Y,B,X	PULL USTACK Y,B,X;
Branch conditional	BCC Location	IF C = 0 THEN GOTO location;
	BNE Location	IF ACC > 0 THEN GOTO location;
	BHI Location	IF HIGH THEN GOTO location;
	BHS Location	IF HIGH/SAME THEN GOTO location;
	BLE Location	IF ACC < = 0 THEN GOTO location;
Unconditional branch	BRA Location	GOTO location;
	JMP B,X	GOTO MEM(B + X);
	JMP [B,X]	GOTO [MEM(B + X)];
Jump subroutine	JSR Location	JUMP SUB;
Return from subroutine	RTS	RETURN FROM SUB;
Return from interrupt	RTI	RETURN FROM INTERRUPT;

M. F. Smith is manager of technical strategy and research for Istel (formerly BL Systems) in the UK and is also a visiting fellow at the University of Reading. From 1980 to 1984 he was a lecturer in microprocessors at the University of Reading, and from 1978 to 1980 he was manager of computer applications for Petroconsultants SA, Genova. Smith received a BSc in geology from University College Wales, Aberystwyth, UK, in 1972, an MA in micropaleontology from California State University, Fresno, in 1973, and a DPhil in oceanographic instrumentation from University College Galway, Ireland, in 1980.

Mark Sealey is with Plessey Radar, Cowes, Isle of Wight, UK. He has been involved in the computer simulation of several types of radar systems and is currently working with naval radar. Sealey received his BSc in computer science from the University of Reading in 1983. During his studies, he was sponsored by Plessey.

Questions about this article can be directed to Smith at Istel Limited, Grosvenor House, Redditch, Worcestershire, UK.

Yigal Hoffner has been a member of the Computational Sciences Research Group at the Department of Computer Science, University of Reading, since 1983, where he has built a statically reconfigurable multimicroprocessor system which is being used to solve partial differential equations. From 1980 to 1983, he was a research assistant in the university's Microprocessor Unit, specializing in microprocessor applications. His work included the design and construction of microprocessor-based data acquisition and control systems. Hoffner received a BSc in computer science and cybernetics at the University of Reading in 1980.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 162 Medium 163 Low 164

MicroStandards

Editor: Robert G. Stewart/Stewart Research Enterprises/1658 Belvoir Drive/Los Altos, CA 94022

Report on the Paris Multibus II meeting

and some thoughts about the future of the committee

by Hubert Kirrmann
Brown, Boveri Research Center
Switzerland

The IEEE Computer Society standardization committee on Multibus II held a meeting in Paris on June 3 and 4 of this year. At attendance were 14 persons, four from the host company (Bull), three from Intel, two from French companies (Matra and PTT), two from Germany (Siemens and AEG), and two from Poland, in addition of the reporter, who acted as chairman. There were no representatives of backplane manufacturers. Only one attendee had experience in building a (prototype) Multibus II system. The attendees came with an expectant attitude—to learn the state of Multibus II, to explain and defend the current draft, or to learn about the design of the bus.

There was accordingly little interaction between the attendees and the organizers. Only the representative of AEG had prepared concrete items to discuss. Most newcomers had not read the draft in detail.

Nor were there any disputed items. The previous meeting in Baden had already made clear that the committee wanted to accept the bus as it is, especially the requirement set up previously of "interoperability with the systems designed in 1984," which closes the way to any further changes. Even persons who recognized the validity of some changes preferred to back Intel so as not to delay the standardization process.

The only open items now are the definition of the interconnect space (the draft only specifies a few registers out of 512) and the mechanisms for unsolicited message passing, which are very superficially defined in the specification.

Maurice Hubert of Bull presented a list of six requirements for the interconnect space, but offered no solutions. Intel, by contrast, presented its solutions for the organization of the interconnect space it had designed into its Control and Service Module (CSM) and memory modules. The Intel implementation is heavily dependent on the use of a microcontroller which augments the real estate of the board but presents some problems for simple boards because of the mixing of RAM and ROM in the same 512 bytes.

The board's real estate was also discussed. The bus interface logic, even highly integrated into silicon, takes slightly less than one-fourth the available board space. This is why NCR and Central Data advocate using triple-Eurocards for their MB II products and expect the committee to approve their way. The committee in Paris just resolved not to prohibit the triple-Eurocard implementations, but still considers the double size (6U) as standard, in an effort to reduce the number of options.

The new mechanical standard modular order system (MOS), submitted by Siemens at the IEC, was again proposed for consideration. This new standard is completely metric, while the current Eurocard mixes English and metric measures. This complicates CAD/CAM aspects. The committee felt that it was premature to discuss the topic.

"Solicited message passing" was also discussed. Solicited message passing was renamed "message passing," while unsolicited message passing was renamed "interrupt message." It appears that Intel's concept on message passing is quite weak at this time. Flow control, OSI layering, and datagram/virtual circuits aspects are completely ignored in the present draft. It seems that it would be best in the future to make a separate standard out of this part. The same can be said of the serial bus, which has no clear position now in the architecture (but is not part of the standardization work).

The committee reviewed the current draft for terminology and unclear points and delivered numerous valuable remarks to the draft editor, Scott Tetrick of Intel. Most discussion revolved around the terminology, especially the use of the terms "requester," "responder," and the hierarchy of transmission "cycle," "sequence," "transfer," "conversation," etc.

Retrospective on the committee's work

The success of the committee's work can only be measured with regard to its objectives. Intel's interest is to gain the widest market share for its silicon and systems with Multibus II. The user community is interested in a widely used standard which suits their needs. Therefore, there should not be a conflict in interest, as long as Intel's marketing strategy is to keep an open system.

The committee's role is therefore to improve the present draft to augment its quality and acceptance. Two kinds of improvements are possible: in the design itself and in its description. While the committee can act as proofreaders of the draft and make a quantity of changes in the wording of the draft, the leeway for technical change is very small.

Intel's attitude can be described by the lemma: "We make the standard, we don't follow it." The Intel people have done a careful analysis of the bus and, for every proposal for technical changes, the Intel experts had good reasons at hand why their sôlution was correct and the proposal could not be accepted. Few arguments were influenced by the marketing, however. But even if a proposed change were meaningful, it is doubtful whether it would be accepted by Intel.

Technical changes between Revisions B and C were not the result of the committee's work, but came from Intel. Proposed changes were ignored by Intel until it was too late to incorporate them.

When technical changes are proposed (such as permitting 21 slots), Intel expects somebody else to prove their validity, although it is clear that the Intel engineers will not accept a change that they themselves did not check.

The "no change" attitude is understandable, since a certain hysteresis is needed and desired for stability in the draft and confidence of the users in the product. However, the committee was only offered a chance to intervene when the design was already frozen.

Intel's interest is that the Computer Society rubber-stamp its design as soon as possible. To put it correctly, the same applies to all committees which standardize existing designs. The success of this operation would of course enhance the marketing position of Multibus II, which is starting slowly at the moment. Everybody is expecting his neighbor to start first on Multibus II, but the bus is gaining momentum.

The committee is wise enough not to make changes against Intel's will. The members—mostly from industry—are very conscious that any change that is not blessed by Intel can delay and jeopardize the market chances of the bus. Under these conditions it is also understandable that none in the committee will take to heart the task of defining the interconnect space, for instance. By the time it is finished, Intel will come out with its own proposal and impose it.

One can argue that no changes are needed since MB II fits exactly the requirements set up in the objectives list. The point is that the objectives list is an *a posteriori* specification. As a result of the committee's attitude, it has been drafted to suit the MB II specification rather than the reverse. But let's be fair: one can always specify the problem in such a way as to imply the solution.

Nevertheless, my feeling is that the bus would have come out better if the following points would had been considered:

32-bit optimized. Deleting the 16-bit justification would allow interoperability between all cards and, by suppressing an option, would favor future technologies over 16-bit, and would give an edge over VME. The 16-bit optimization reflects Intel's market analysis. (Will the 386 be delivered only after 1990?)

Broadcast. This permits caches and replicated global memories, which increase the bus's average bandwidth by about 500 percent. Intel's justification is that Multibus II is not "common-memory" oriented, but "message" oriented. I wonder whether we need a 10MHz parallel bus to transmit messages when the average sending time within the kernel is in the order of 200 μs. This is a typical job for a serial bus (a 10 Mb/s serial bus transmits a 100 bit message in 10 μs). Also, the "message passing" philosophy results from an Intel choice regarding iRMX 86 and iMMX. The users interested in real-time application could have difficulties living with that philosophy, although little is known about it until now. The binding of the bus with Intel's operating systems is therefore a major consideration to prospective users.

Fixed geographical address which supplies the initial arbitration ID. This would spare the microcontroller on the Central Service Module (CSM) at the expense of three pins which could be taken from one ground and two +5V lines. Intel's argument that 45W per slot as required does not seem reasonable).

Moving the CSM to the left side of the backplane. The present position is in the middle of the backplane. This position changes depending on the length of the backplane used.

I recognize that it is too late to incorporate these changes now in the draft (except the last), and I'm not even sure that they would increase the acceptance of the bus. But the real question is what the contribution of the committee can be. The committee members find it difficult to educate themselves sufficiently on the subject, and have no clear picture of what they can contribute to the standardization process except by way of revising the wording of the Intel documents.

State of the standardization work

The Multibus II committee is working practically on its own. It has, until now, not received a Project Authorization Request (PAR) from the IEEE, which means that its work is not an official IEEE project. The IEEE set up the P896 (FutureBus) activity in 1978 in order to have a single 32-bit bus standard. Intel had the opportunity to collaborate in this committee but chose to develop its own product (but used some ideas of the P896 design). Intel was offered a second opportunity to develop a common bus in April 1983. At that time, a week-long meeting was held at Wilsonville, Oregon (not far from Aloha) and an effort was undertaken to merge NuBus (TI), Multibus II, and FutureBus under the P896 label. The operation failed due to lack of flexibility in all participants, including the P896 working group, which was divided between the roles of referee and player. Now the IEEE Computer Society is confronted with six 32-bit buses: FastBus, FutureBus, Multibus II, NuBus, Versabus, and the two-connector VME (VME-32), and can choose to standardize them all, none, or only its legitimate child FutureBus. [Ed.Note—FastBus is already an adopted IEEE standard].

To justify a separate standardization effort, the candidate buses must differ from one another in a significant way. Multibus II can be measured with respect to its nearest competitors using the Eurocard format: Future Bus and VME bus.

FutureBus (P896) was formed by the Computer Society to develop an independent 32-bit bus before anyone else. The open discussion in the committee created a abundance of new ideas and advanced the state of the art – but the discussions delayed the draft some years, and irritated the industry. FutureBus' main result is its fertilization of other projects (VME, Multibus II, NuBus), which are now its descendants. As it stands now, FutureBus is technically a good design, but it is still a long way from a commercial reality. [Ed. Note—Textronix is building systems to P896]. FutureBus will be more costly than Multibus II (real estate taken by the interface, uncoventional drivers, buried layer backplane, triple Eurocard) and so it will belong to a higher price (and performance) class. It is manufacturer-independent, but not supported by a processor manufacturer until now and its commercial future is uncertain.

VME is gaining momentum especially because of its connection with the 680XX processors. It is based on Eurocards and has about the same useful card size as MB II: although MB II has larger boards, the difference is compensated by the facts that (1) the real estate taken by a MB II interface is larger than for the VME interface and (2) MB II requries a Command and Service Module. It has about the same

useful speed as MB II. VME-32 has a design flaw which restricts its utility in 32-bit systems. Like MB II, it is 16-bit oriented, but it is not capable of managing correctly unaligned quads, doublets and 24-bit entities. Even the 68020 has inteface problems with it. Furthermore, the binding between VME and 680XX is strong. One can hardly speak of "manufacturer independence." This is also true from MB II, but to a lesser extent.

For this reason, the MB II users are in need of a bus which would support their Intel, National, or DEC line of processor and offer about the same performance as VME. The difference is less at the physical level than at the level of system philosophy, which has implications up to the kernel and to the development environment.

In summary, all 32-bit buses cover the same range of applications, are more or less equally adapted for multiprocessors, and run at speeds that vary according to the benchmark. They do not differ much in technique, but in marketing. Clearly, something went wrong with the standardization process.

Outlook

Should the IEEE Computer Society continue to support the Multibus II activity?

The interest of a professional society is to obtain a standard for a 32-bit bus that is manufacturer-independent and suits the need of a broad community of users. It is not in IEEE's interest to standardize four or five 32-bit buses. Definitively, one can argue that FutureBus is enough.

I tend however to favor a PAR for Mutlibus II for the following reasons:

1) It is better to have more than one standard than to have no standard at all. This will at least prevent others from making new buses.
2) Although MB II is not very different technically from its competitors, it is different in terms of system philosophy and support organizations.
3) If VME is standardized (P1014), then the non-680XX users should be offered a similar opportunity to balance the market.

It is clear that this argumentation imples that NuBus should be standardized, too, if TI wishes so.

Finally, I would urge that a clear policy be followed with respect to collaboration with manufacturers in the future. If the committee is reduced to a mere proofreader of documents, it will be difficult to find motivated and qualified people to attend the meetings. But if the committee sets up requirements and designs and standardizes at the same time, the draft may never be finished. A professional society will be most eager to standardize a work with which it can identify itself, not one it feels has been imposed upon it. This means that the Computer Society should be involved in the requirement stage or even in the design and not placed before a *de facto* document. The Computer Society must be able to bring different manufacturers to the same table and obtain a common solution; that is, it must stick to its referee role. I think a mechanism must be created to instill mutual confidence that the standardization will not unduly delay the commercialization and that the result will conform to the objectives set up by the Society for the benefit of its members.

32-bit Buses—six now and some more to come

Some history. . .

When the Computer Society started the standardization work on 16-bit buses (S-100 and Multibus), it became apparent that these buses could have been significantly improved, but that it was too late to make significant changes. In fact, the changes made to the S-100 during the standardization process perhaps harmed its market, although they improved its technical qualities undoubtedly. To avoid repeating such a situation, the Microprocessor Standard Committee set up in 1978 the P896 "FutureBus" standardization committee on 32-bit backplane buses, at a time when only FastBus (MIM) and some mainframe buses like SBI existed. Since then a lot of work has been done; there were numerous technical discussions until all newcomers understood the issues before they began arguing them. The interaction of participants from the USA and Europe was tense at times, but finally the big options were settled: the bus would be built on Eurocards, be handshaken, use 100mA drivers, and be optimized for 32-bit on one 96-pin connector, with no options at all. Since the committee depends on volunteer work and cannot work at the pace of industry, this process took time, and the need for a 32-bit bus became urgent in the industry. Some manufacturers then started to develop their own buses, and took ideas from the P896 work. At least one company tried to market "P896" boards which they developed from an early P896 specification. So, VME was created in 1981 by Motorola's participant in the P896 committee. It was followed by NuBus (the synchronous version of Western Digital/Texas Instruments adapted from the handshaken one of MIT),and Multibus II in 1983. National Semicondtor and Zilog (ZBI) had also a design ready, but dropped it. Finally, the committee came up with the final P896 specification in 1984, so the situation we face in 1985 is that there are four 32-bit buses competing for an IEEE standardization, to which one should add FastEus and Versabus. Some more are on the horizon, like DEC's BI.

Standardization means reducing options and developing wide acceptance. The dilemma of standardization is that a general-purpose solution in normally technically inferior to a specialized solution, so there is always a reason why one should not follow the standard and instead make one's own bus (apart from the fact that it's fun). The standardization of several designs is only justified if they differ from one another sufficiently.

In order to compare the four 32-bit buses based on Eurocards which currently are being, or which have applied for a standardization by the Computer Society, we list their technical data in Table 1. Fastbus and Versabus are not listed because the Fastbus is a special design used in nuclear experimentation and Versabus has been superceded by VME-32.

On the system philosophy

All these four buses claim to support multiprocessors. However, here some definition is needed, since today every computer is a multiprocessor containing several processors for disk access, graphic display, etc. Let us distinguish three general classes of multiprocessor systems:

- A *centralized* multiprocessor, in which one processor runs the kernel and treats the other as slaves. This processor has access to the local memory spaces of the slaves.

Table 1.
Major 32-bit buses.

	FutureBus	Multibus II	NuBus	VME32
Support	IEEE-CS	Intel	TI (MIT)	Motorola, Philips, Mostek
System Optimization	32-bit federalist multiprocessor	16-bit autonomous multiprocessor with central service module	32-bit federalist multiprocessor clock module	16-bit centralized multiprocessor; service module optional
Address Bits	32	32	32	32
Address Spaces	MEM, CSR	MEM, message, I/O, intercon.	MEM	6 address modifier bits
Data Width	8,16,24,32	8,16,24,32	8,16,24,32	8,16,32
Parity	mandatory	mandatory	optional	none
Optimized For: (justified)	32-bit	16-bit	32-bit	16-bit
Width Steering	individual byte strobe	address + size	address + size	address + size + byte strobe within doublet
Sequential Transfer	yes, unbounded	yes, unbounded	yes, bounded	yes, unbounded
Geographical Address	5 bits, wired	T-pin, needs central service module	5 bits, wired	none
Arbitration	decentralized	decentralized, but needs CSM to initialize	decentralized	daisy chain, 4-level
Interrupt	memory-mapped or serial bus	message-space or serial bus	memory-mapped	7 interrupt levels or destinations or serial bus
Protocol	3-way handshake	synchron 10MHz	synchron 10MHz	handshaken
Transfer mode:	multiplex	multiplex	multiplex	simplex
Broadcast	yes	no	no	no
Technology	100mA OC special drivers	48mA 3S 64mA 3S TTL	48mA 3S 60mA 3S TTL	48mA 3S TTL
Max Speed[1] (single-read transfer)	200 Mb/s	106 Mb/s	106 Mb/s	152 Mb/s
Board Size	triple * 280 1024 cm^2 (preferred)	double * 220 512 cm^2 (preferred)	triple * 280 1024 cm^2	double * 160 mm 373 cm^2
Connectors	1 DIN	1 DIN	1 DIN	2 DIN
Number of Slots	32	20	16	19
Auxiliary Buses	serial bus (undefined)	execution iLBX serial iSSB	none	execution VMX serial VMS
Vintage	1985	1983	1983	1981
Status	experimental	few boards 2 vendors	only used in NuMachine	about 6 vendors (most VME boards are VME-16)

1. Maximum estimated speed, based only on time spent in the bus protocol, but not in the logic or in the memory access time. Single read transfer was chosen because it is the most frequent transfer; sequential transfers increase the throughput of all buses and reduce the advantage of the simplex bus VME. A simplex bus is by nature 30 percent faster than a multiplex bus, but only for single transfers.

- A *federalist* multiprocessor, in which all processors own a copy of the kernel and communicate over a shared memory, using a shared memory space, but also have a private memory space.
- An *autonomous* multiprocessor, in which the processors do not share a common address space but communicate by messages, as in a network. They only have the name space in common.

In our case, does the term "multiprocessor" mean *federalist* multiprocessor? The following criteria express more or less what is needed to support it:

- Decentralized arbitration with a "fairness" strategy guaranteeing that every module will receive access to the bus within a bounded time.
- Decentralized interrupt system which allows every module to send an interrupt to all other modules or to a group of destinations.
- Locking mechanism for semaphore operations.
- Initialization line which allows a defined start-up of the system (reset-not-complete).
- A broadcast mechanism which allows implementing a replicated global memory and speeds up considerably the access to shared data.

VME for instance does not support such an architecture very well, principally because its interrupt system is very close to that of a monoprocessor interrupt system. There are only seven possible destinations for an interrupt, but this could be fixed by using the VMS serial bus for interrupts.

Multibus II, at the other extreme, allows a federalist multiprocessor operation, but supports primarily an autonomous multiprocessor. When a board wants to send a message to another, it sends a request for buffer space; the other board acknowledges that a buffer of the correct size has been allocated, and the first board starts transmission. A message-passing system allows communication between boards without a common address space; it is, in this sense, half a network. This method is rather inefficient in terms of communication response time, since the kernel is involved to allocate memory. Shared variables make a far better use of the bus. On the other hand, this method is comfortable when the address space of the processors is limited. So Multibus II is practically a backplane local-area network.

NuBus and P896 do not specify the communication mechanism, but it is assumed that it is by shared variables. The system aspects are not well defined and work is in progress for it in the System Architecture Study Group chaired by Paul Borrill.

On the cost

The cost of a bus connection depends principally on the real estate taken by the interface. In addition, one must consider the real estate taken by the central services board like a CSM or a clock module, and divide it by the average number of boards. The price of the backplane (some cost $500 US) must also be divided evenly to get an estimate. The P896 backplane is the dearest, since it uses a buried track technique. Multibus II needs a six-layer backplane while both NuBus and VME need at least a four-layer backplane.

The real estate taken by the interface is the major cost factor. We show next how it can be estimated. The real estate depends principally on the numbers of ICs necessary to drive the bus and to execute the protocol conversion. We shall consider only the address and data path circuitry (buffers, latches, registers, decoders) since the protocol control cost is essentially constant for all configurations and can easily be cast in silicon. We estimate that any buffering or bidirectional driver requires a package for each byte.

Take the simple processor that accesses a simplex bus like VME. Suppose the bus and processor have 32-bit address and 32-bit data. The processor is capable of becoming bus master, but the bus cannot access the processor card (except through an interrupt). The interface consists of about eight octal bus drivers for the address and data path and two control line drivers. The signals of the processor are directly used to steer the bus lines, no protocol conversion is required, and the cost is minimal (see Figure 1).

When the bus is multiplexed, the interface must multiplex address and data. The number of bus drivers drops since there are less bus lines to drive, but more routing packages are needed to buffer the address and the data. The interface needs only four drivers but eight octal routers/buffers, as well as a more complicated control logic (Figure 2).

Consider now that the bus is used in a multiprocessor configuration in which each processor board is accessible from the global bus, either to address a local memory, a local configuration PROM (interconnect space in Multibus II), or to transmit memory-mapped interrupts. Therefore, the board must have both the master and the slave logic on it. The data routing could be made as in Figure 3.

The local bus constantly reflects what happens on the global bus, so that the slave address decoder can decode the traffic on the global bus independently from the processor's work. An independent access of the slave address decoder to the bus would be feasible, but when the bus is multiplexed there is in addition a local bus, and using it reduces the system bus load.

The above solution presents a problem because the local memory is addressed in DMA mode. When there are two processors in the system, and each accesses each other's local memory at the same time, a deadlock results. Two solutions are possible:

- First, break the deadlock by "rolling back" one processor. This is only possible when the processor is capable of interrupting an instruction and resuming it later, i.e., when the processor is fitted to work with a virtual memory (68010, 32032, . . .). Most ancient processors are not capable of this, so this is not a general solution.
- Second, prevent the deadlock by introducing an additional level of buffering, as shown in Figure 4. The full interface then costs about 20 packages. A similar configuration is required for Multibus II when it has a local memory on board. The question is whether it is meaningful to have a local memory at all, not only because of the additional buffering, but also because of programming problems (the same location exists under 2 different addresses, variables should be copied anyway for communication, etc.). Everybody puts local memories on board, but it seems that the utility is small in a multiprocessor system.

The price keeps on rising if the 32-bit

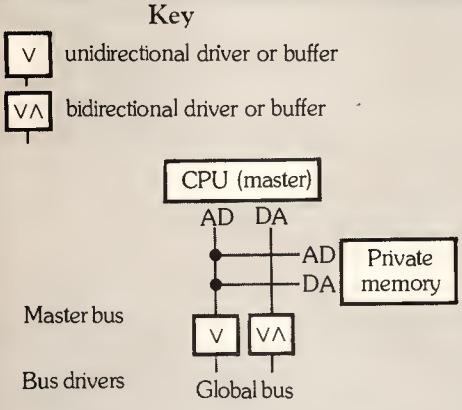


Figure 1. Simplex bus, master only. Eight packages.

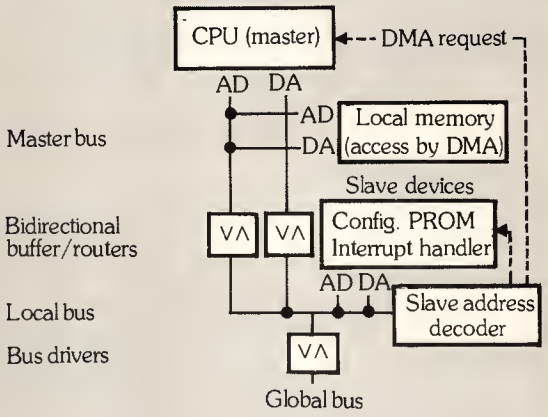


Figure 3. Multiplexed bus, master and slave circuits. 16 packages.

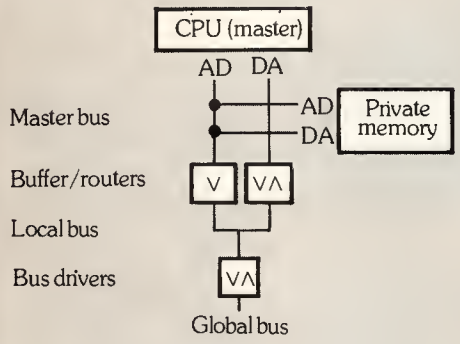


Figure 2. Multiplexed bus, master only. 12 packages.

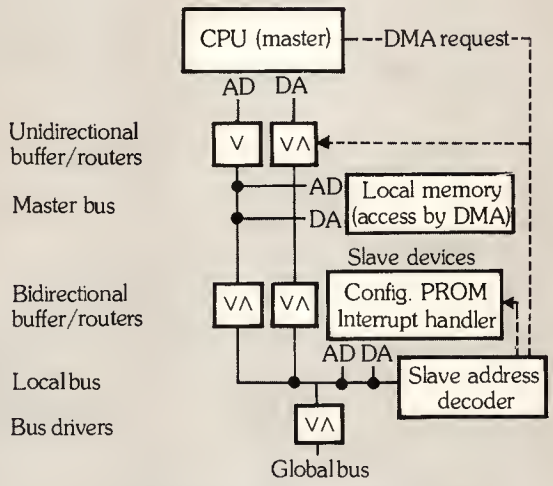


Figure 4. Multiplexed bus, buffered local memory access. 20 packages.

bus is 16-bit optimized. Then, four additional buffers/routers are required to transfer the 16-bit entities to the other byte lanes when they are transmitted on the two higher lanes, so the total increases to 24 packages.

Note that the interface would cost more if the bus would be simplex: there would be some four more bus drivers because a local bus is required in any case. Therefore, a multiplexed bus is more efficient when a multiprocessor configuration is envisioned, as in Multibus II. However, if only a centralized multiprocessor is envisioned, as in VME, the cost is lowest when the bus is simplex (Figure 1).

Conclusion: the VME boards will be cheaper than the Multibus II boards in single processor configuration, but more costly in a multiprocessor configuration with a local memory.

Comments

VME. The oldest 32-bit bus, VME was designed from the 16-bit VME version by adding a second connector. Unfortunately, the designers did not provide individual strobes for all four-byte lanes, so it is now impossible to make 24-bit transfers. Transfers of 24-bit entities are required to transmit 32-bit words which are aligned at an odd memory location (as VAX does), by transmitting first one byte and then the remaining three bytes. Even the 68020 has problems with the VME-32 protocol. The designers should have spared some of their too numerous arbitration or interrupt lines for this. Furthermore, a system problem arises with VME-32: Since most lines on both connectors are taken for the parallel transfer of address and data, few lines remain for the local execution bus VMX, which has to be multiplexed to fit in the remaining 64 lines. This leads to the paradoxical situation that the system bus is faster than the local execution bus. In this light, it is clear that VME is not a federalist bus, but a centralized bus. Most users of VME do not use it in multiprocessor configurations, but as a processor bus. Finally, the compatibility between

continued on page 89

MicroReview

Editor: David L. Hannum/AT&T Information Systems

A show?

by David L. Hannum

This month, a little something different: a story about an emerging technology and the aura and problems surrounding it.

The technology is videotex. In the beginning, it had the simple, elegant goal of bringing information retrieval to the masses.

But things have changed. In late June, I had the chance to attend the Videotex '85 show in New York City. Many of the large computer manufacturers (DEC, IBM, AT&T, etc.) were represented there; but unlike Comdex, NCC, and others, it was very much a show for the videotex elite, not for the applications-oriented individual or end-user.

To understand the depth of this metamorphosis, you have to understand first what videotex is, and where it came from. A little history please. Back in 1972, the videotex/teletext idea was born in the mind and on the paper of one Sam Fedida in Britain. Fedida developed a scheme, which he called "viewdata," to encode and decode text and graphics transmitted over a twisted-pair network (telephone lines). This information retrieval system, simple to implement and provide, could use inexpensive and readily available equipment to generate images based on the ASCII character set plus.

Fedida's scheme quickly fell victim, however, to both improved transmission technology and lowered cost for sophisticated logic within the terminals. This lead to much hair pulling and upheaval in the industry until AT&T emerged with a new approach to the transmission scheme, a protocol termed the North American Presentation Level Syntax, or NAPLPS. This protocol, accepted as a standard by ANSI and the CSA, led to superior graphics capabilities; but the industry was still plagued by the fact that users could implement the system only on dedicated terminals that were generally too dumb and not cheap enough. As a result, few applications folks would spend money to receive the service in their businesses or homes. And, as the very disappointing Videotex '85 show testifies, marketing efforts on behalf of this technology still lack a focus on the actual needs and circumstances of users.

Now that dumb terminals have given way to personal computers with some add-on boards, and graphics and text transmission are superior to anything previously seen on the market, why has videotex not yet penetrated more deeply into our daily lives? Why is home shopping for both soft goods and hard goods not a reality today; why are we not buying our cars or selling our products or labor from the comfort of our living rooms; why are we not conducting financial transactions, being schooled, or making use of any other number of

"Can we communicate with machines as effectively as we can with other humans?"

applications available with this technology; why are these things not reality now?

The truth is that, for some of us, some of these applications are a present reality; within the next few years, many of them will be in limited operation throughout the country. Does this mean that we are ready to let this next-generation capability change our lives and our interactions with those people that have traditionally provided us with goods and services? Can we communicate with machines as effectively as we can with other humans?

We are beginning find some answers in user response to existing forms of videotex technology. The Commodity News Service (CNS) has been established as an information retrieval service for commodity traders, and a similar system is in place for discount brokers; CitiBank is advertising its online banking service; Epcot Center uses videotex to help visitors find their way and retrieve information about exhibits; Dow Jones has an online financial market service (information only); Compuserve and Source exist as rudimentary examples, used more for play and computer information than for serious information management; and there are others. Why then did the show dwell on the form and not the content, or (to borrow a phrase) the medium and not the message?

It seems to me that this is so largely because the purveyors of the technology are not the builders of the applications, and this base of builders is still so small that it is overwhelmed by the purveyors (who still run the show, so to speak).

My hope is that the videotex industry will begin to develop and implement useful applications that will turn this exciting technology into an indispensable tool for the public. This segment of the information-management-and-movement industry offers the most exciting marketing potential available to the industry leader who stands up to point the way, even if that leader ends up being a coalition or industry advisory group such as ours.

It is my perception that videotex can and will chart our next great step toward the goal of Equal Access to Information; it will provide many interactive services that exist now only as the figment of someone's imagination. We may move as quickly into this age of armchair access to the world, traveling through the use of videotex and large screens, doing our shopping and handling our finances at home, as we did to jump from the 64K chip to the 256K chip. It may take some of us some time to readjust.

In what year will you be reading this column on your videotex screen? I hope it will be soon.

In the next issue: IC design packages for the IBM PC.

MicroStandards *continued from page 87*

VME-I6 (one connector) and VME-32 (two connectors) is a rather difficult story in practice. Configuration problems are also expected because of the different address width and the numerous options.

VME-32 is clearly the cheapest bus as long as no protocol translation takes place (that is, as long as 680XX processors are used), and as long as the processor boards have no dual-ported local memory accessible from the outside. VME is supported by a number of chips from Motorola, Signetics and Mostek.

NuBus and Multibus II. These two buses are so little different that it is unbelievable that the designers could not agree on a common design. Multibus II has a better system approach (interconnect space, reset-not-complete, serial bus), while NuBus is streamlined for a 32-bit design. Both buses lack the capability to make broadcast transfers, which prevents the use of caches on the bus. Multibus II is hampered by the necessity of a Central Service Module which is in principle only needed for initialization and for the clock. It may also have problems because of its two levels of compatibility, 16-bit and 32-bit data. The commercial future of NuBus is uncertain, while Intel is pushing Multibus II and is developing bus controllers for it. Several Multibus II boards have been introduced based on 80286, 32032 and J-11.

FutureBus. FutureBus (P896) has a very consistent 32-bit design but is difficult to manage. It offers the highest theoretical speed but requires a costly technology. In fact, the difference between NIM's FastBus (based on ECL) and P896 is small. The newly designed bus drivers of P896 (and now available from National Semiconductor) are in fact a rediscovery of the virtues of the small voltage switching range of ECL. Another problem is the lack of support from the industry. Ferranti announced bus controllers for FutureBus and Tektronix is introducing a system based on it, but no other designs are yet reported. Finally, there is no system support in sight for FutureBus.

Outlook

As things stand now, the 32-bit bus market will be dominated by the struggle of VME-32 versus Multibus II. Simple users of 680XX will prefer the VME-32, while Intel, National, and DEC processors will fit better on Multibus II in a sophisticated multiprocessor configuration. But users should not neglect the system aspects: who is going to deliver the software and the development tools to run these multiprocessors? The bus is only the tip of the iceberg.

MicroLaw

by Richard H. Stern/Law Offices of Richard H. Stern/2101 L Street NW, Suite 800/Washington, DC 20037

Further chip rights developments

Things are happening so fast in the semiconductor chip topography protection field that each month's new developments seem to eclipse last month's. The US Copyright Office, which is charged under the Semiconductor Chip Protection Act (SCPA) with the obligation of registering chip layouts, has issued new regulations for chip topography registration. (See *Federal Register,* Vol. 50, No. 125, June 28, 1985, pp. 26714-21.) In large part the new regulations back down from the intransigent position that we criticized in this column in the June issue. But on some points the Copyright Office refuses to bow to semiconductor industry objections.

The Copyright Office still thinks that a cell from a cell library (and, by the same token, a macro for a gate or linear array) is an "intermediate form" of a chip,[1] and that its protection is *therefore* dangerous to the public interest, even if the legislative history of the SCPA indicates congressional desire to protect cells. Although the Copyright Office (in an excess of misplaced self-confidence) feels that its basic concept is sound, it is willing to bow slightly under the impact of semiconductor industry criticism.

The 20 percent rule. The Copyright Office has decided to somewhat soften the infamous "20 percent rule." Readers will recall that, under the earlier version of this rule, no cell could be registered unless it constituted at least 20 percent of a complete integrated circuit. Since cells amounting to more than 20 percent of an IC are extremely rare, this rule meant that most cells were unregistrable. The Office will now allow cells to be registered without regard to their percentage contribution to complete ICs, subject to two important limitations.

- **Deposits.** When a cell "represents less than 20 percent of the area of the intended final form, a visually perceptible representation of the work which reveals the *totality* of the mask work contribution to a person trained in the state of the art" must be deposited with the Copyright Office to identify the work. Applicants are no longer required to deposit die exemplars; this is now optional for cells.
- **Cells actually used.** If the cell has been embodied in actual complete ICs, the IC rather than the cell must be registered and deposited. If the cell proprietor cannot register the complete form of the IC because he lacks control over that form, a statement must be made to that effect on the application (Form MW).

The first limitation means that ion implantation information or other trade secret data revealed by some masks cannot be withheld in the case of cells, as it can for complete ICs. It is unclear whether that is a serious commercial problem. The second limitation apparently means that the Copyright Office will not register a cell (as a cell alone) that the owner has already commercially exploited by incorporating it into a complete IC. It is possible that the Copyright Office also means that a cell cannot be registered as such if it has been embodied in silicon in any form that includes more than the cell itself.[2] In any event, this rule indicates that a new cell or macro should be registered at once, if it is to be registered at all. The application should be filed before the cell is licensed or made available to customers and before the proprietor uses it in his own ICs. This may be a nuisance, but it is not as bad as the previous position of the Copyright Office, discussed last month.

Arrays. The original Copyright Office position on the registration of arrays, such as gate arrays or linear arrays, appeared to be that they could not be registered in unpersonalized (unmetallized) form if they had ever been personalized, even for testing. Apparently, the Copyright Office has not yielded on this point. It explains, "The Copyright Office sees no compelling reason why applicants should be allowed indiscriminately to select earlier versions of intermediate forms when they have in their possession more complete versions." Of course, no one wants to see array manufacturers engaging in indiscriminate selection of what they send to the Copyright Office to be registered and protected against copying by pirates, or at least no one dares to argue in favor of it, so doubtless the industry will have to acquiesce in the Office's position.

One array manufacturer that I know has a solution that may be useful to others. (He has asserted no proprietary right over this device.) He has his new designs for linear arrays made up for him by a silicon foundry, without metallization. After he receives the wafers from the foundry and gives them some preliminary tests, he sends the wafers back to the foundry for metallization at the same time that he files his Form MW for the array with the Copyright Office. He feels that his designs are final enough at that point that at most the layout will need a few minor "tweaks" after he checks the metallized version. He says

that if he is ever wrong about that, he is willing to pay the Copyright Office another $20 filing fee and spend another $50 or $100 worth of his time to file a new application for registration of the next version of the array. We invite reader comment on whether this will work for everybody. Circle number 135 on the Reader Interest Card if you feel that this approach has some general merit; circle number 136 if you feel that it is unworkable except in select circumstances. We will report the results of the survey in the next issue.

Trade secrets. The Copyright Office will now allow more masks to be withheld to preserve trade secrets. Originally, only two masks could be withheld; the Copyright Office based this figure on the conception that many ICs would be made from mask sets of approximately five masks. (By way of comparison, the MC68020 microprocessor chip has about 24 masks.) Now, the Copyright Office will allow two masks to be withheld out of each five. In addition, it is now permissible to block out part of a mask. When a mask is withheld, the information on it is to be identified by an alternative means. These include microfiches of printouts of database types (a practice favored by IBM) and overlays or partial composites for the withheld masks, where less than 50 percent of the total area has been blocked out.

Defective chips. The original regulations and the present version require deposit of four die exemplars as identifying material for commercial chips, along with composites, overlays, or other visually perceptible material. The Copyright Office has become aware, however, that people were submitting "red dot" chips as die exemplars. On the first day of registration, a representative of one well-known semiconductor manufacturer told me that his sink was pretty red after he finished preparing his deposit material for the ceremony. For some reason, many people do not want to give usable chips away to the Copyright Office.

The Copyright Office has bowed to this practice. It states that it will accept defective chips "provided that the mask work contribution would be revealed in reverse dissection of the chips." Apparently, that means that a chip is acceptable if the defect does not obscure the identity of what is claimed. Actually, the Copyright Office has no way of determining whether a die satisfies this rule. The registrant just acts at his own peril. If in mask work infringement litigation the die is so bad that the court does not think that the defendant's chip is a copy of the plaintiff's deposited chip (when the defendant's expert gets finished analyzing it), then the plaintiff will lose the case. That is not very likely. In all probability, sinks may safely run red.

International protections

The other news in this field is that foreign nations are joining the US in protecting IC topography. Section 914 of the SCPA provides that citizens of a foreign nation may register their IC layouts and enjoy protection in the US against chip piracy if (1) the foreign nation is making reasonable progress toward establishing a system of chip protection "on substantially the same basis" as the SCPA (which applies to US chips) and (2) if the nationals of the foreign nation are not engaging in piracy of US chips. The US Patent and Trademark Office is responsible for determining whether these conditions are met. If it decides that they are met, it issues an order under section 914 allowing citizens of the foreign nation to register their IC layouts. The order may be revoked if conditions change, and it may be superseded by a Presidential Proclamation that permanent reciprocal protection should be granted to the foreign nation's firms. However, whether or not the order is revoked, extended, or made permanent by a Presidential Proclamation, all registrations made while the order was in effect will remain in force for the same ten years that a registration under the SCPA by a US firm continues in effect.

Foreign nations are joining the US in protecting IC topography.

The Japanese Diet has passed "An Act Concerning the Circuit Layout of a Semiconductor Integrated Circuit." The Japanese law in the main parallels the SCPA, and permits US owners of semiconductor layouts to register their ICs for protection in Japan. The Japanese Ministry of International Trade and Industry (MITI) is now preparing regulations necessary for the implementation of the new law, which is expected to become operational before the end of 1985. In response, the US Patent and Trademark Office has issued an order under section 914 of the SCPA, authorizing Japanese semiconductor firms to register their ICs with the US Copyright Office for full protection under the US law. The order is to be reviewed in a year to determine whether implementation of the Japanese law has given US firms fully reciprocal protection.

Sweden advised the US that it plans to pass a chip protection law, and the US Patent and Trademark Office has issued a similar order under section 914 in favor of Swedish manufacturers. Canada advised the US of similar plans, and received a similar order.

The Netherlands, the United Kingdom, and Australia each advised the Patent and Trademark Office that they believe that their present general copyright laws (passed in 1912, 1956, and 1968, respectively) already protect IC topography, even though the laws do not mention ICs and no case involving ICs has ever been decided under them. However, the Patent and Trademark Office felt that it was important to encourage these nations to protect ICs. It therefore issued a one-year order giving Netherlands and Australian firms reciprocity of protection, and a two-and-one-half year order for UK firms.

There are significant problems with the British law, however, as the Semiconductor Industry Association (SIA) pointed out to the Patent and Trademark Office. The problems center on whether UK chip protection will really be "on substantially the same basis" as the chip protection of the SCPA. Because of these problems, the SIA recommended that the Patent and Trademark Office initially issue only a several-month order and urged thorough review of the matter during that period.

First, it is unclear whether and how the UK copyright law really applies to ICs. It is uncertain that a microscopic, virtually invisible design is copyrightable under UK law, particularly when the product is sealed in an opaque container so that it cannot be seen. Second, it is unclear whether the UK is about to change its copyright law. The European Community has complained that the UK copyright law creates barriers to intra-European Community trade. And the House of Lords has decided to review the latest decision in the field, in which British Leyland, an automobile manufacturer, stopped competition in the sale of spare tailpipes for its cars as a copyright infringement.[3] A governmental commission of experts recommended that Parliament repeal the present UK copyright law applicable to industrial designs; and some observers believe that the House of Lords at least is ready to follow this recommendation. The SIA's main concern, of course, is merely that US firms might be promised IC protection only to lose it later.

A further problem exists with the UK copyright law's treatment of reverse engineering and innocent infringement. In the recent British Leyland tailpipe case, to be reviewed by the House of Lords later this year, the defendant had reverse-engineered the tailpipes from the dimensions of the underside of the car. The defendant claimed that this was not copyright infringement but "fair use" of the plaintiff's copyright as to the tailpipes. The UK Court of Appeals said that this kind of reverse engineering was *not* defensible as fair use. Yet, if the facts were changed from tailpipes to ICs and the case occurred in the US, the defendant's conduct would probably be considered legitimate reverse engineering, expressly permitted under section 906(a) of the SCPA.[4]

In a different but related vein, the Patent and Trademark Office has questioned whether reciprocal protection should be given to a country, such as Sweden, that perhaps "overprotects" ICs, in the sense of giving IC proprietors protection far beyond that of the SCPA. Without its express legitimation of reverse-engineering, the SCPA probably could not have passed. The US law reflects a balance and compromise between the respective interests of IC proprietors, their competitors, IC users, and the public. If a foreign country strikes a very different balance of these interests, its law may not satisfy the SCPA's reciprocity requirement, which demands that the foreign law give US firms protection "on substantially the same basis" as the SCPA. The UK copyright law seems to tilt the balance much further to the side of the first chip seller, and against the reverse-engineering competitor, than the US SCPA does. It might well be that if a US firm engaged in second-sourcing of another firm's chip, it would be found innocent of liability in the US because of the reverse-engineering defense, but guilty of copyright infringement in the UK because the UK does not recognize that defense. (The result would be the same irrespective of whether the plaintiff firm were a US or UK semiconductor manufacturer.)

The SCPA reflects a balance between the interests of IC proprietors, their competitors, users, and the public.

A similar, but probably less grave, question may be raised by the innocent infringement defense, which the US SCPA recognizes and foreign copyright laws do not. Under the innocent infringement defense, an equipment manufacturer is not liable for innocently incorporating infringing ICs in its equipment and reselling them. At most, the unknowing equipment manufacturer is liable for a reasonable royalty on ICs that it bought without knowledge and resold after it learned of the IC topography rights. Another question is whether copyright law's 50-75 year term of protection is excessive for ICs.

Finally, the Commission of the European Communities is preparing a draft directive to its member states on IC protection. If the directive becomes final, it will require all EEC states to protect IC topography, either by a special (*sui generis*) law as in the US and Japan, or by copyright. At this time, the Patent and Trademark Office has not yet been able to complete its review of the EEC petition.

References

1. An "intermediate form" of a semiconductor chip product is the first m layers fabricated in the manufacturing process of a chip where there are n masking (photolithography) steps and where $0 < m < n$. For example, a wafer with an oxide coating into which windows have been opened for doping is an intermediate form of the semiconductor chip product. So, too, is an unpersonalized (unmetallized) array or unprogrammed mask-programmable ROM.

2. If the Copyright Office means the latter, too, then it may be that no cell is registrable as such if it has ever been placed on a wafer with other circuitry for testing.

3. Under UK copyright law, manufacture and sale of a product can be an infringement of the copyright in the blueprints or other technical drawings for the product. That the competitor never saw the blueprints is irrelevant. The copyright law in the US and most other countries is to the opposite effect.

4. Alternatively, the second-sourcing might be defended on the ground that the function of the product dictated its layout. Under the SCPA, layout features dictated by function are not infringing when copied. This rule probably extends to features needed to make a chip "form, fit, and function compatible," such as pin layout.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 183 Medium 184 Low 185

New Products

Editor: Kenneth Majithia/IBM Corporation

Supermicro accommodates up to 20 users

Altos Computer Systems has introduced the Altos 2086, a supermicrocomputer that is said to accommodate up to 20 users and runs the Xenix 3.0 operating system.

Featuring a 16/32-bit 8-MHz Intel 80286 central processing unit and a modular architecture, the supermicro is designed for sale to end users in medium-sized businesses and specific vertical markets.

According to the company, the Altos 2086 can be used in two ways—either as a standalone, general-purpose supermicrocomputer or as a node in a distributed network. The system can be networked with other Intel-based Altos products via WorkNet, Altos' local-area network. With Altos PC Path attached to WorkNet, the 2086 can act as a file server and communications gateway for IBM-compatible personal computers. In addition, the Altos 2086 is software-compatible with the IBM-PC/AT.

Other communications products—such as 3270 Bisynch, 2780 Bisynch and 3270 SNA/SDLC—allow the Altos 2086 to communicate with mainframes. Using the standard X.25 communications protocol, the computer can also gain access to public data networks.

The base configuration of the Altos 2086 sells for $19,990 and includes 2M bytes of RAM, an 80M-byte hard disk, a 1.2M-byte floppy disk drive, a 60M-byte streaming tape unit, and an Altos III terminal. RAM can be expanded to 8M bytes in 2M- or 4M-byte increments. Hard disks can be upgraded to 189M bytes (formatted) in 63M-byte increments. The modular design includes eight board slots, three of which are open.

The Xenix 3.0 operating system available for the 2086 supports user programs of up to 1M byte. An optional Intel 80287 floating point processor chip is available for the 2086.

Altos Computer Systems is located at 2641 Orchard Parkway, San Jose, CA 95134; (408) 946-6700.

Reader Service Number 5

Software development system uses PC host

Micro/sys, Inc. has introduced the Bus/Bridge family of software development packages for ROM-based, board-level embedded systems. Target hardware environments include Multibus (8085 and 8086), VMEbus (68000), STD Bus (Z80 and 8088), and Multibus II (80286). In each case, the IBM PC is used as the host environment from which target system software is developed and debugged.

In addition to these target environments, the designer can choose from three software systems: 1) the target processor's assembly language, 2) one of the compilers available for the IBM PC (if the target system uses an iAPX86-family processor), or 3) a proprietary assembler and compiler. Each Bus/Bridge package includes a target system CPU board, target system RAM, a ROM monitor to operate the target system, an assembler for the target processor, an IBM PC diskette with target communication and file conversion utiliies, and an RS-232 cable for connection to an IBM PC. The designer adds a target system card rack with power supply and an IBM PC. Other packages add language compilers to the basic hardware/software components. Packages run from $1295 for STD Bus Z80 development, $1995 for Multibus development, $4795 for VMEbus 68000 development, or $5595 for Multibus II 80286 development.

Micro/sys is located at 1101 Grand Central Avenue, Glendale, CA 91201.

Reader Service Number 6

The Micro/sys Bus/Bridge software development system is available for target hardware operating in Multibus, VMEbus, STD Bus, and Multibus II environments

PC CAE system supports color graphics

FutureNet Corporation has announced the Dash-3C color version of the Dash PC-based electronic engineering CAE workstation.

According to FuterNet, the system permits a design engineer to assign a preferred color or to use the system's predefined default colors to identify four separate classes or groups of graphic display elements. These include symbol interconnection lines, symbols, symbol cell boundaries, alphanumeric fields, and command menu; drawing and menu cursors (except pin cursors), command line, menu headings, area definition lines, symbol definition instructions, disk directory, and library directory; MODE status field value, message line, tagged objects, alphanumeric insert cursor, symbol definition target lines, fast pan windows, rubber banded lines, and direct lines; and full scale windows, display border, and principal status field headings and values.

The complete Dash-3C system includes an IBM PC, XT, or AT computer system with minimum 256K RAM; an IBM enhanced color display and enhanced graphics adapter board with graphics memory expansion card; IBM keyboard; MS/DOS operating system software; FutureNet mouse and parallel port board, with cable and modular plug; C.Itoh 15-inch Prowriter II printer with cable; and FutureNet Dash-3C software and user manuals.

The IBM PC, XT, and AT-based Dash workstation is available either as a turnkey system or as an add-on package for those who already have the IBM computer with color display. The Dash line also includes postprocessors for documentation tasks, such as net list, list of materials, and design check; and STRIDES, a hierarchical design program.

FutureNet notes that the Dash system offers a range of CAD interface translators to other systems such as Applicon, ComputerVision, Racal-Redac, Sci-Cards, Spice and Tegas.

The Dash-3C add-on package, Model D3-MAP, is priced at $5980. The price of the complete Dash-3C system is from $10,880 to $14,380, depending on the IBM computer included.

FutureNet is located at 9310 Topanga Canyon Boulevard, Chatsworth, CA 91303-5728; (818) 700-0691

Reader Service Number 8

CHMOS microcontroller offers 16-MHz speed

Intel Corporation has introduced a high-speed version of its 80C51BH CHMOS 8-bit microcontroller. The chip, called the 80C51BH-1, has maximum operating frequency of 16 MHz, which is said to be 25 percent faster than the 80C51BH and the 8051 HMOS microcontroller.

The chip combines CHMOS technology with a built-in Boolean processor for bit-level data manipulation, 32 programmable I/O ports, programmable power modes, and a UART port. The 80C51BH-1 operates at voltages from 4 volts to 6 volts, and has an operating current of only 20.5 milliamperes at 5 volts and 16 MHz. In the power-down mode, current is less than 50 microamperes at voltages of 2 volts to 6 volts. Operating at full speed, the 80C51BH-1 can support a serial data-transfer rate of up to 500K bits per second.

The 80C51BH-1 is designed for applications in computer peripherals (as a controller in such devices as disk drives, tape drives and line printers), in telecommunications (as a controller in modems and digital line cards), and in high-performance industrial process control.

The 80C51BH-1 has 4K bytes of ROM and 128 bytes of RAM on-chip, together with the ability to address a total of 64K bytes each of external data and program memory. The 80C51BH-1 is compatible with the 8051 and other MCS-51 chips. It is also available in a ROMless version, 80C31BH-1. The component is available at a unit price of $11.55 in quantities of 5000.

Intel is located at 5000 West Williams Field Road, Chandler, AZ 85224; (602) 961-2756.

Reader Service Number 7

Multibus communications board adaptable to custom protocols

Systech Corporation has introduced its DCP-8804 Data Communications Processor, a 0.5M-byte computer on a board designed to "provide serial interfaces with a wide range of standard and custom communication protocols," according to company representatives.

A four-channel, IEEE-796-compatible, high-performance general-purpose communications processor on a single printed-circuit board, the DCP-8804 is said to raise system throughput by offloading communications-related functions from the main computer.

The company also provides OEM customers with Unix-compatible software drivers to permit them to adapt the DCP-8804 to their computers.

According to Systech, a Multibus computer system, with the addition of a DCP-8804 controller and appropriate firmware, can be configured to function as a remote concentrator, nodal processor, front-end processor, or communications gateway. Systech's DCP enables Multibus-based computers to operate in asynchronous, bisynchronous, SDLC, HDLC, X.25 and SNA communications protocols.

Other DCP-8804 hardware features include:

- Sockets for installation of up to 64K bytes of EPROM;
- 4 multiprotocol serial communication channels, employing Zilog 8530 SCC chips; all serial channels can be configured for RS-232C/RS-422A/RS-449 operation;

Supermicro operates on Multibus

Matrox Electronic Systems has announced a 32-bit processor board for Multibus I featuring the National 32032 CPU, 32082 MMU and 32081 FPU chip set. An optional chip set can also be installed.

The NAP-2000 single board computer and MB-2000 2M-byte companion memory board are said to offer 32-bit supermini computer performance comparable to the Vax 11/780 for under $8000. Using the dedicated 32-bit memory expansion bus, MX-Bus, the NAP-2000 can access up to 16M bytes of memory including a high-speed 64K-byte cache. New and existing Multibus designs can be upgraded to use the supermicro.

The NAP-2000 processor board features the 10-MHz 32032 CPU, a 32081 FPU coprocessor for high speed floating point arithmetic, and the 32082 Memory Management Unit. Additional on-board resources include up to 512K bytes of ROM, 128K bytes of zero-wait-state RAM, a real-time clock with battery backup, 16 levels of vectored interrupts, two RS-232 serial ports and bus interfaces to the Multibus and the 32-bit Matrox MX-Bus. An EPROM-based real-time executive and monitor are optional.

With a second set of CPU, MMU and FPU, the performance of the NAP-2000 is said to increase 70 percent.

The MB-2000 is a 2M-byte memory board with an 8K-byte cache, error detection and correction, and transparent memory scrubbing. The MB-2000 is triple-ported to Multibus, iLBX bus, and the 32-bit MX-Bus.

The MB-2000's 8K-byte cache memory is claimed to provide zero-wait-state access for 16-MHz processors on the MX-Bus with a hit ratio of 90 percent. Also provided is the provision for external battery back-up protection. The MX-Bus has been developed to provide a 32-bit data path for 32-bit processors on Multibus. The address and data lines are nonmultiplexed and can provide data rates up to 40M bytes per second. The MX-Bus can allow the NAP-2000 to access up to 8 MB-2000s, thereby providing up to 16M bytes of directly accessable memory with a 64K-byte cache. To facilitate short access times, there are no arbitration cycles on the MX-Bus; it is an extension of the local CPU bus. The 10-MHz NAP-2000 can access memory on the MB-2000 via the MX-Bus with zero wait-states. Physically, the MX-Bus is two 50-line ribbon cables on right angle connectors on top of the Multibus cards. The MX-Bus specifications have been proposed to IEEE for inclusion in the IEEE-796 standard as a 32-bit extension to the Multibus. Copies are available on request.

The NAP-2000 is priced at $3995 in single quantities. The MB-2000 is also priced at $3995. OEM discounts are available.

Matrox Electronics is located at 1055 St. Regis Boulevard, Dorval, Quebec, Canada, H9P 2T4; (514) 685-2630.

Reader Service Number 9

- 2 8237A-5 DMA (Direct Memory Access) controllers providing 8 dedicated DMA channels for full-duplex DMA operation on the four serial channels;
- an 80188 CPU providing two more high-speed DMA channels for full-duplex DMA operation to and from the host Multibus memory.

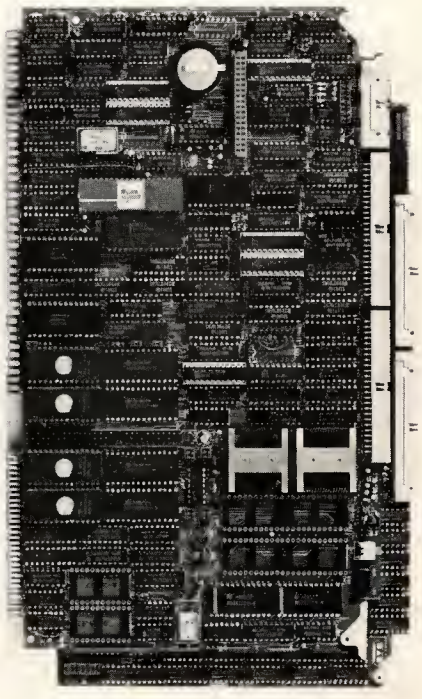
The DCP-8804 contains two user-programmable counter/timers and a "channel attention" function that can be mapped to reside in either Multibus memory or I/O space. Five LEDs allow for visual verification of board status. The DCP-8804's Multibus "lock" feature is supported in both dual-ported and host Multibus memory operations.

Data can be transferred to and from the host system by using the DCP's 16K-byte, dual-ported memory buffer or the 24-bit full-duplex DMA circuitry. The dual-port buffer is jumper-selectable to reside in any contiguous 16K-byte boundary of the host Multibus system's memory map.

The communications processor is offered with a real-time, multitasking executive and with Unix-compatible device drivers. With 256K bytes of RAM, the DCP-8804 sells for under $2000 in OEM quantities of 100 or more. X.25 optional software is priced at $7000 with OEM discounts available. The 3780 emulation software's initial installation package costs $4500; royalties are $95 per unit in OEM quantities.

Systech is located at 6465 Nancy Ridge Drive, San Diego, CA 92121; (619) 453-8970.

Reader Service Number 35



The NAP-2000 single-board computer from Matrox Electronics runs on Multibus I.

Laser optic system designed for PCs

Reference Technology, Inc., has introduced the Clasix DataDrive Series 500 laser optic information distribution system, based on CD ROM technology and designed for users of IBM personal computers.

The Clasix DataDrive Series 500 is a desktop read-only laser optic peripheral designed to deliver large databases on prerecorded media to users of IBM and compatible personal computers. Attachment to other minicomputers can be accommodated through the use of the company's hardware and software modules. The Clasix CD is a 4¾-inch (12cm) Philips-Sony CD Rom-compatible prerecorded optical disc capable of delivering up to 550M bytes of user data or 250,000 pages of textual information. Reference Technology's Tridecc data preparation services support the production of Clasix CD discs. In addition, the company's STA/F File software package standardizes the access and format of large volumes of data published on read-only optical discs, thereby allowing databases to be used on various systems, independent of operating environments. The software also extends the addressing range of IBM PC-DOS to enable standard PC applications to access databases up to 4M bytes in size.

The Clasix DataDrive Series 500 is priced at $1535, including hardware to attach to the IBM PC product line (PC, PC-XT, PC-AT). STA/F File software for use with the Series 500 is priced at $110 per user license. Tridecc data preparation services for the Clasix CD involve a basic fee of $8000 per disc side plus $250 per additional reel of input data tape (beyond the first). Clasix CD replicas are priced at $15 per copy.

Reference Technology is located at 1832 North 55th Street, Boulder, CO 80301; (303) 449-4157.

Reader Service Number 10

Reference Technology's Clasix DataDrive Series 500 laser optic peripheral is designed to deliver large databases of reference material to IBM PC users.

Floating point array processor operates on Multibus

Mercury Computer Systems, Inc. has introduced the ZIP 3232 8- and 16-Mflop array processors, which employ the AMD 29325 VLSI chip. The processor is a three-board set for Q-bus and Multibus, supported on Sun, Intel, and Motorola systems as well as the microVAX II.

According to the company, the product is targeted at the OEM market for signal, image, and scientific processing. The ZIP 3232 is priced at about $940 per Mflop in single unit prices and $610 per Mflop in OEM quantities.

The AMD 29325 performs both addition and multiplication in the same device and so eliminates pipelining. Typical performance for the ZIP 3232 is 2.8 ms for a 1024-point complex FFT, 279 ms for a 3×3 convolution on a 512×512 image, 1.9 μs/output point for a 16 tap FIR filter, and less than .75 sec for a 2 dimensional FFT on a 512×512 image.

The ZIP 3232 features the same architecture and programming environment as the ZIP 3216, Mercury's 16-and 32-bit block floating point coprocessor. The control processor is based on the AMD29116, and memory is 128K bytes expandable to 16M bytes. The programming environment features ZIP/C, a C-like language, and off-line development tools which permit program writing, debugging, and benchmarking on systems including the VAX, 68000-based systems, and the IBM PC. The 16-Mflop version of the 3232 costs about $9750 in OEM quantities and the 8-Mflop version is about $7500. Memory can be expanded in 2M-byte increments for $3900 and 512K-byte increments for $1950.

Mercury Computer Systems is located at 600 Suffolk Street, Wannalancit Technology Center, Lowell, MA 01854; (617) 458-3100.

Reader Service Number 11

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 186 Medium 187 Low 188

Product Summary

Editor: Kenneth Majithia/IBM Corporation

For more information, circle the appropriate Reader Service Number on the Reader Service Card at the back of the magazine.

Manufacturer	Model	Comments	Rs No.
Chips/Components			
Intech, Inc. 2270 Martin Avenue Santa Clara, CA 95050-2781 (408) 988-4930	RGB DAC 3400 family digital/ analog converters	CMOS, triple, video speed D/A converters provide interface between digital circuitry and analog inputs to monitor, provide composite sync and blanking signal levels for CRT beam sweep circuits, offer 4096-color palette. Models 3400, 3404, 3405, 3408 provide varying levels of internal RAM to store colors from palette. Priced in 1000-unit quantities from $39.95 to $79.95, depending on model.	21
Micro Power Systems 3100 Alfred Street Santa Clara, CA 95054 (408) 727-5350	MP7684 series A/D flash converter	8-bit component, developed using molybdenum gate CMOS process, features 20MHz sampling rate. Can achieve 9-bit resolution by connecting two in series, 40MHz speed by connecting two in parallel. Available in 28-pin CERDIP package from $46.60 to $275.00, depending on model.	22
Communications			
Server Technology, Inc. 1095 East Duane Street Sunnyvale, CA 94086 (800) 835-1515	EasyLAN local-area network	Package connects IBM PCs or compatibles for under $100 a connection. Network commands parallel PC-DOS commands; can be configured with PBX and modem connections. Password scheme limits access to LAN. Requires PC-DOS 2.0, 128K RAM. Two-user kit, including software, manual, and 30-foot cable, available for $179.95; expansion kit for $109.95. Software, cables also priced separately.	23
Softronics, Inc. 3639 New Getwell Road Suite 10 Memphis, TN 38118 (901) 683-6850	Softerm communications software	Product comes in versions compatible with the IBM PC, Tandy Model 2000, and Apple II, IIc, and IIe. Features 24 terminal emulations, including emulation to permit Tandy 2000 to be used as a console in Tandy Model 16 Xenix system. Said to integrate concurrent communications with IBM PC-compatible programs. Available retail or from manufacturer from $135 to $195, depending on model.	24
Peripherals			
Kurzweil Applied Intelligence 411 Waverly Oaks Road Waltham, MA 02154 (617) 893-5151	KVS 3000 Voicesystem speech recognizer	Programmable device compares spoken input with 3000 token or sample utterances to provide recognition of 1000 words or phrases in speaker-dependent mode or several hundred words in speaker-independent mode. Recognition takes place at a rate of 250 msec per word. Features 640K RAM. Available as board set for Multibus host systems for $5000 in single quantities, $3000 in OEM quantities, or as self-contained unit for $6000, $6500 with IBM PC interface. Additional memory boards available.	25
MicroPhonics Technology 234 SW 43rd Street, Suite B Renton, WA 98057 (206) 251-9009	Pronounce speech input system	Designed for IBM PCs and compatibles, system permits user to define vocabulary files of up to 128 words each for compatible software to execute command sets of up to 255 keystrokes or to standardize unrelated programs under voice control. Consists of board, microphone, software with predefined vocabularies for Wordstar and Lotus 1-2-3. $895.	26
Tandon Corporation 20320 Prairie Street Chatsworth, CA 91311 (818) 993-6644	TM362 3½-inch Winchester drive	Half-height drive measures 1⅝ inches high, 4 inches wide, 5⅞ inches long, has 20M-byte formatted storage capacity with average access time of 80 msec, features pseudo-closed-loop head-positioning system. $300 in OEM quantities.	27

MicroCourses

Editor: James J. Farrell III

IEEE Micro accepts announcements of short-course listings pertaining to microprocessor and system design. Announcements should designate course titles, locations, dates, costs, and contact address and telephone number. To be of greatest use to our audience, information should be received at the address below at least **two months before** cover date, and should be current at least **two months after** cover date. Send announcements to MicroCourses, Dept. M-SC, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

Ada Programming Series, 90-hour PC-based training course; *Programmable Controller Fundamentals,* 11-hour PC-based training course; on-site implementation services available. **Contact: Jacque Mihm, Control Data Corporation; (612) 853-2705.**

Data Communications System Components; Network Design, Operations, and Management; Network Protocols and Standards; SNA; LANs; Digital PBXs; X.25 and Packet Switching Networks; $750-1350. **Contact: Systems Technology Forum, 9000 Fern Park Drive, Burke, VA 22015; (800) 336-7409.**

Micros for Managers: Software, eight videotape lectures available for lease or purchase. **Contact: Engineering Renewal and Growth, Colorado State University, Fort Collins, CO 80523; (800) 525-4950.**

Personal Computer and STD Computer Interfacing for Scientific Instrument Automation, September 19-21, Greensboro, N.C.; $450. **Contact: CEC, Virginia Polytechnic Institute, Blacksburg, VA 24061; (703) 961-4848.**

Database Management and Fourth Generation Languages for Personal Computers; Data Communications: Network Design, Integration, and Applications; Data Communications and Networking for the IBM PC and Other Personal Computers; Fourth Generation Data Management Software; Structured Systems Development Using Fourth Generation Languages; Information Systems Architecture; Decision Support Systems; Micro - Mainframe Links; courses held through September in various US locations; $695-895. **Contact: Software Institute of America, 8 Windsor Street, Andover, MA 01810; (617) 470-3880.**

Microprocessor Fundamentals; Microprocessor Troubleshooting; Microprocessor Software, Hardware, and Interfacing; 16-Bit Microprocessors; Software Engineering for Micro and Minicomputer Systems; Real-Time Software Design; Database Management Systems; Digital Image Processing; Digital Signal Processing; Designing Real-Time Hardware for Digital Signal Processing; Digital Control Systems; Modern Pattern Recognition Systems; courses held through November in various US locations; $965-995. Videotape training series, self-study courses, and on-site courses also available. **Contact: Integrated Computer Systems, 6305 Arizona Place, Los Angeles, CA 90045; (800) 421-8166 outside CA; (800) 352-8251 inside CA.**

Networking the IBM PC; Data Communications Systems; Office Automation; Selecting a PBX System; Unix/Xenix; Network Communications Protocols; SNA; LANs; $695-745. **Contact: Center for Advanced Professional Education, 1820 East Garry Street, Suite 110, Santa Ana, CA 92705; (714) 261-0240.**

Artificial Intelligence: An Applications-Oriented Approach, September 5-6; *Fundamentals of Data Communications,* September 9-11; *Modern Digital Communications,* September 16-18; *Operating Systems for Microcomputers,* September 16-20; *Electronic Computer Printing,* September 18-20; *Expert Systems: A Practical Application of Artificial Intelligence,* September 23-24; *Local Area Data Communications,* September 25-27; *Modern Electronic Interconnection and Packaging Systems,* September 26-27; *Practical Techniques for Artificial Intelligence Programming,* September 30-October 2; *Digital PBX: Integrated Voice/Data,* October 3-4; *Decision Support Systems,* October 7-9; *Writing Professional and Technical Communications,* October 7-9; *Magnetic Recording Materials,* October 8-10; *Advanced Professional and Technical Writing,* October 10-11; *Digital Magnetic Recording,* October 11-12; *Digital Telephony,* October 21-25; *Systems Analysis Techniques,* November 14-15; *Electronic Reliability Screening,* November 18-21; *Modern Communications and Signal Processing,* November 18-22; *Digital Transmissions Systems Engineering,* December 2-6; $650-920. **Contact: Continuing Engineering Education, George Washington University, Washington, DC 20052; (800) 424-9773.**

Introduction to Unix; Text Processing in Unix; Shell Programming; Using C with Ultrix-11; Using C with Ultrix-32; self-paced instruction programs; on-site lectures/labs also available. **Contact: Digital Equipment Corporation, Educational Services, 12 Crosby Drive BUO/E55-54, Bedford, MA 01730; (800) 332-5656, ext. 005.**

Fiber-Optic System Testing Seminar, one-hour videotape available for purchase or rental. **Contact: Fotec Inc., 529 Main Street, Box 246, Boston, MA 02129; (617) 241-7810.**

DOS Expert, computer-enhanced training course; $295. **Contact: ATS, 21250 Califa Street, Suite 107, Woodland Hills, CA 91367; (800) 426-8737.**

VLSI Packaging, September 9-11; *Test Procedures for Precision Instrumentation and ATE Systems,* November 5-6; *Power Semiconductor Devices,* December 5. **Contact: Public Information Division, National Bureau of Standards, Gaithersburg, MD 20899; (301) 921-2721.**

Data Communications Series, Database Management Series, Software Engineering Series, Personal Computers Series, Information Resource Management Series, Artificial Intelligence Series, courses held in Rockville, Maryland, and in various US locations; $585-995. **Contact: Institute for Advanced Technology, Control Data Corporation, 6003 Executive Boulevard, Rockville, MD 20852; (800) 638-6590.**

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 192 Medium 193 Low 194

Letters *continued from page 4*

This program uses just 10 instructions. It occupies in total 20 bytes of code and executes in 32 clock cycles, if one assumes a 50 percent chance for the data byte to have an even parity.

Further improvement in speed (clock cycles per data byte) can be achieved if one applies A. Perez' algorithm to a word of data. The code for a look-up program of the type proposed by A. Perez would appear as in Figure 3. This code uses seven instructions. It occupies 16 bytes plus 512 bytes for the look-up table. It executes in 37 clock cycles, five cycles more than the previous "on the fly" program.

There is finally no advantage in using a look-up program on the 8086/8088. Even the argument that one could change the CRC polynomial by changing the look-up table values and keeping the same code does not stand. With reference to R. Keir's "To the Editor" note in the same April issue, one does not use the same initial CRC values with the SDLC (X**16+X**12+X**5+1) polynomial as with the CRC-16 polynomial. Nor are the CRC words stored the same way with the data.

Applying the A. Perez algorithm to the SDLC polynomial, we can show that the CRC is formed by modulo 2 addition of the four words listed in Figure 4. This can be translated into the program listed in Figure 5. This program needs 15 instructions. It occupies in total 30 bytes and executes in 38 clock cycles, one more than the look-up program.

Even by optimizing the assembler-generated code from D.V. Shouse's Fortran program, one would not be able to easily create a code as compact and fast as presented above. So why not keep bit-manipulating algorithms in assembler.

Ivar Kjelberg
Centre de Recherche en Physique
des Plasmas
Ecole Polytechnique Fédérale
de Lausanne

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 189 Medium 190 Low 191

Author's response:

I again agree that there are drawbacks in using Fortran programming for bit manipulations. As I mentioned in my article, modifying the parity bit tests, the carry bit tests, and the AND-OR instructions into XOR instructions will reduce the number of bytes from the generated code. Also, by eliminating all references to memory, that is, using only the registers (as done by Mr. Kjelberg) one can further reduce the generated code. This reduced program would be better to use. However, my purpose was to show how to implement A. Perez' idea in a high-level language, like Fortran and to show the generated code.

D. V. Shouse, Sr.
General Railway Signal
Rochester, NY

```
3        32 D0          XOR  DI,AL                    ;form X
2        8A DA          MOV  BL,DL                    ;copy X
3        32 FF          XOR  BH,BH                    ;clear BH
2        D1 E3          SAL  BX                       ;form word offset
2        8A D6          MOV  DL,DH                    ;copy high byte of CRC
3        32 F6          XOR  DH,DH                    ;clear DH
13+9     31 97    0000  XOR  DX,[LOOKTBL+BX]          ;look-up
-------------
37 cycles
```

Figure 3. Assembler code for a CRC look-up program as proposed by A. Perez.

```
MSbit                                                                    LSbit
|                                                                          |
X8  X7  X6  X5  X4  X3  X2  X1  C16 C15 C14 C13 C12 C11 C10 C9
0   0   0   0   0   X8  X7  X6  X5  X4  X3  X2  X1  0   0   0
X4  X3  X2  X1  0   0   0   0   0   0   0   0   X8  X7  X6  X5
0   0   0   0   0   X4  X3  X2  X1  0   0   0   X4  X3  X2  X1
```

Figure 4.

```
3        32 C2          XOR  AL,DL          ;form X
2        8A D6          MOV  DL,DH          ;high byte of CRC
2        8A F0          MOV  DH,AL          ;save X
2        8A F8          MOV  BH,AL          ;form second word
3        32 DB          XOR  BL,BL          ;clear BL
2        D1 CB          ROR  BX
3        32 D8          XOR  BL,AL          ;form third word
2        D1 CB          ROR  BX
2        D1 CB          ROR  BX
2        D1 CB          ROR  BX
4        24 0F          AND  AL,0F          ;mask out X8...X5
3        32 F8          XOR  BH,AL          ;form last word
2        D1 CB          ROR  BX
3        32 D8          XOR  BL,AL          ;finish last word
3        33 D3          XOR  DX,BX          ;form new CRC
-------------
38 cycles

Input:   old CRC in DX, data byte in AL.
Output:  new CRC in DX, old values of AL and BX are lost.
```

Figure 5.

Access

Recent articles on microcomputing

Editor: Peter R. Rony/Department of Chemical Engineering/University of Delaware /Newark, DE 19716

Articles

Business Computer Systems

Vol. 4, No. 5, May 1985:

D. W. Post, "Upgrading Your Micro," pp. 76-85.

S. Austin, "New Technology Printers: Options for (Almost) Everyone," pp. 99-119.

R. W. Ridington, Jr., "Topview: Waiting for the Software to Make It Sing," pp. 121-129.

J. Heid, "Microsoft File: Leader of the Macintosh Database Pack," pp. 129-136.

Vol. 4, No. 6, June 1985:

S. Austin, "Mixed Grades for Training Software," pp. 89-100.

J. Heid, "Face Off: Mac vs. IBM PC in a War of Words," pp. 101-103.

S. W. Bryan, "SPSS/PC Does Almost Anything with Statistics," pp. 107-112.

Byte

Vol. 10, No. 5, May 1985:

G. Williams, "The AT&T Unix PC," pp. 98-105.

A. L. Schumer, "Set Extensions with Apple Pascal," pp. 128-135.

B. Webster and T. Yonkman, "Methods: A Preliminary Look," pp. 152-155.

C. Macie, "Smalltalk-PC," pp. 155-160.

J. Anderson and B. Fishman, "The Smalltalk Programming Language," pp. 160-165.

R. Karjewski, "Multiprocessing: An Overview," pp. 171-181.

G. D. Beals, "Extending Microprocessor Architectures," pp. 185-198.

W. G. Paseman, "Applying Data Flow in the Real World," pp. 201-214.

J. E. Roskos and C. Hsieh, "Data Movement Primitives," pp. 239-252.

G. M. Vose, "Software Review: True Basic," pp. 279-288.

Vol. 10, No. 6, June 1985:

J. Lawler, P. Hairsine, and A. E. Miller, "Interactive Audio in a Videodisc System," pp. 108-117.

S. D. Fenster and L. E. Ford, "SALT," pp. 147-164.

P. Robinson, "The SUM: An AI Coprocessor," pp. 169-180.

D. Ushijima, "Inside AppleTalk," pp. 185-200.

M. Fichtelman, "The Expert Mechanic," pp. 205-216.

W. F. Grunbaum, "SWITCH," pp. 221-226.

A. Huston, "Structuring Basic," pp. 243-262.

W. P. Stevens, "Using Data Flow for Application Development," pp. 267-276.

L. Ledbetter and B. Cox, "Software ICs," pp. 307-316.

Computer

Vol. 18, No. 4, Apr. 1985:

M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto, "Requirements-Based Testing of Real-Time Systems: Modeling for Testability," pp. 71-79.

Vol. 18, No. 5, May 1985:

G. H. Sockut, "A Framework for Logical-Level Changes Within Database Systems," pp. 9-27.

J. DeRosa, R. Glackemeyer, and T. Knight, "Design and Implementation of the VAX 8600 Pipeline," pp. 38-48.

M. D. Abrams, "Observations on Operating a Local Area Network," pp. 51-65.

Vol. 18, No. 6, June 1985:

D. D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems," pp. 9-27.

P. C. Patton, "Microprocessors: Architecture and Applications," pp. 29-40.

P. B. Schneck et al., "Parallel Processor Programs in the Federal Government," pp. 43-56.

K. Hwang, "Multiprocessor Supercomputers for Science/Engineering Applications," pp. 57-73.

K. Murakami, T. Kakuta, R. Onai, and N. Ito, "Research on Parallel Machine Architecture for Fifth-Generation Computer Systems," pp. 76-91.

Computer Law & Practice

Vol. 1, No. 3, Jan./Feb. 1985:

A. Schulkins, "Software Copyright," pp. 100-101.

Datamation

Vol. 31, No. 9, May 1, 1985:

O. Serlin, "Departmental Computing: A Choice of Strategies," pp. 86-96.

Vol. 31, No. 11, June 1, 1985:

A. Pantages, "Beyond Today's Blue," pp. 22-32.

P. Archbold and J. Verity, "The *Datamation* 100," pp. 37-182.

Vol. 31, No. 12, June 15, 1985:

S. Dittlea, "Befriending the Befuddled," pp. 84-90.

M. Rosenthal and R. Loftin, "PC Software Integration," pp. 95-98.

J. D. Smiddy and L. O. Smiddy, "Caught in the Act," pp. 102-106.

J. Kelly, "Computer Law," pp. 116-126.

Dr. Dobb's Journal

Vol. 10, No. 6, June 1985:

D. Gengle, "Information Age Issues," pp. 52-55.

D. Gay, "C UART Controller," pp. 60-65.

D. Krantz, "Christensen Protocols in C," pp. 66-89.

Vol. 10, No. 7, July 1985:

D. Rindsberg, "The Ultimate Parallel Print Spooler," pp. 46-55.

A. Wilcox, "Designing a Real-Time Clock for the S-100 Bus," pp. 56-90.

R. Duncan, "16-Bit Software Toolbox," pp. 94-109.

Electronics

(The departmental editor welcomes the return of Electronics *(the predecessor* and *successor to* Electronics Week*) with the hope that it will once again become the type of magazine that he enjoyed so much for more than a decade.)*

Vol. 58, No. 24, June 17, 1985:

"Communications: SLIC Chip Shrinks Phone Line Interfaces," pp. 54-57.

Vol. 58, No. 25, June 24, 1985:

J. Young, "IC-Design Automation Strides into Silicon-Compilation Era," pp. 58-63.

"Mass Storage: Erasable Optical Disks Are on the Horizon," pp. 67-69.

Electronics Week

Vol. 58, No. 16, Apr. 22, 1985:

"Memories: One-Megabit EPROMs Invade Disk Territory," pp. 52-54.

H. Bierman and D. M. Weber, "New Display Formats Give Users Better Show for Fewer Bucks," pp. 56-60.

T. Naegele, "Speech Technology Leaves the Realm of Science Fiction," pp. 61-63.

Vol. 58, No. 17, Apr. 29, 1985:

C. Barney, "RISC Technology Moves Off Campus into Commercial Machines," pp. 36-37.

"Sensors: Improved Hall Devices Find New Uses," pp. 59-61.

Vol. 58, No. 18, May 6, 1985:

M. Rand, "Molecular Electronics Research Growing Despite Controversy," pp. 36-43.

R. Rosenberg, "The All-Digital PBXs Stake Their Claim on Office Automation," pp. 57-61.

Vol. 58, No. 19, May 13, 1985:

C. Barney, "ISO Protocols Pose a Dilemma to Potential Users," pp. 34-36.

T. Manuel, "Computers: Parallel Machine Expands Indefinitely," pp. 49-53.

Vol. 58, No. 20, May 20, 1985:

J. Lyman, "MMICs Save Space, Increase Reliability, and Improve Performance," pp. 52-57.

S. Weber, "Custom IC Conference Reflects Significant Gains for System Designers," pp. 61-64.

Vol. 58, No. 22, June 3, 1985:

B. C. Cole, "A Crowd of Hopefuls Warms Up for 32-Bit Microprocessor Race," pp. 50-55.

"Semiconductors: EEPROM Technology Seeds Reprogrammable Logic," pp. 56-57.

Vol. 58, No. 23, June 10, 1985:

A. Wolfe, "Optical Computing Is Beginning to Take On the Glow of Reality," pp. 24-27.

J. Joseph, "Japanese Quit on IBM Software, Turn to Unix," pp. 30-31.

"Speech Systems: Philips's Adapter Box Simplifies Coding, Editing," pp. 44-47.

IEEE Communications Magazine

Vol. 23, No. 4, Apr. 1985:

W. R. Shreve, "Signal Processing Using Surface Acoustic Waves," pp. 6-11.

Vol. 23, No. 5, May 1985:

P. S. Henry, "Introduction to Lightwave Transmission," pp. 12-16.

D. B. Keck, "Fundamentals of Optical Waveguide Fibers," pp. 17-22.

E. E. Basch and T. G. Brown, "Introduction to Coherent Optical Fiber Transmission," pp. 23-30.

C. M. Spierko, "LaserNet—A Fiber Optical Intrastate Network (Planning and Engineering Considerations)," pp. 31-45.

Vol. 23, No. 6, June 1985:

K. Pahlavan, "Wireless Communications for Office Information Networks," pp. 19-27.

S. C. Gupta, R. Visawanathan, and R. Muammar, "Land Mobile Radio Systems—A Tutorial Exposition," pp. 34-45.

IEEE Electrotechnology Review 1984

Articles include "Precision Signal Processing with Switched-Capacitor Techniques," "Local-Area Computer Networks," "Three-Dimensional Computer Vision: Segmenting Scenes into Surfaces," "Computer-Aided Engineering: Tools for the Electronic Engineer," "Automatic Synthesis of Digital Integrated Circuits," "Interactive Videodisc," "Engineering Analysis by Spreadsheet," "Million-Bit Memory Chips," "Supercomputers: A Time of Rapid Change," "The Cosmic Cube," "Of Mice and Windows," "Another Year of the RAM?" "Gallium Arsenide Digital Technology—1984," "Heterostructure Electronics," "Integrating Sensors with Electronics: New Challenges for Silicon," "Can You Trust Your Computer's Numbers?" and "Tunable Semiconductor Lasers."

IEEE Spectrum

Vol. 22, No. 6, June 1985:

R. Bowlby, "The DIP May Take Its Final Bows," pp. 37-42.

D. Botez, "Laser Diodes Are Power-Packed," pp. 43-53.

IEEE Transactions on Education

Vol. E-28, No. 2, May 1985:

D. J. Ahlgren, "Synthesis of a Small Microcomputer—A Project for Undergraduate Laboratories," p. 65-68.

F. DiCesare, S. M. Bunten, and P. M. DeRusso, "Microcomputers for Data Acquisition, Control, and Automation—A Laboratory Course for Pre-engineering Students," pp. 69-75.

J. W. Steadman, R. G. Jacquot, and K. A. Reed, "A 16-Bit Microcomputer for Audio Bandwidth Digital Filtering," pp. 76-78.

IEEE Transactions on Industrial Electronics

Vol. IE-32, No. 2, May 1985:

P. Papasratorn and P. Prapinmongkolkarn, "A Small-Scale Distributed Microprocessor System Using Shared Memory Technique," pp. 97-102.

P. K. Chande and P. C. Sharma, "Microprocessor-Based Flow Measurement System," pp. 103-107.

J. E. Roehl, "A Microprocessor-Controlled Chemical Detection and Alarm System Based on Ion Mobility Spectrometry," pp. 108-113.

N. Chaudhuri, S. Ghosh, and A. M. Ghosh, "A Technique for Simultaneous Measurement with a Microcomputer," pp. 114-119.

R. E. Betz and R. J. Evans, "Microprocessor Control of a Cycloconverter," pp. 120-129.

M. R. Smith, "Interpolation, Differentiation, Data Smoothing, and Least Squares Fit to Data with Decreased Computational Overhead," pp. 135-141.

Macworld

Vol. 2, No. 7, July 1985:

D. Goodman, "Publishing Turns an Electronic Leaf," pp. 70-79.

Microprocessors and Microsystems

Vol. 9, No. 2, Mar. 1985:

J. Chance, "TMS320 Digital Signal Processor Development System," pp. 50-56.

D. Fay, "Interrupts and the Hardware-Software Rendezvous—Microcomputer Software Engineering," pp. 57-63.

S. Pegler, "Gas Burner Control Using Microprocessors," pp. 64-70.

H. A. J. Al-Riahi, "Microcomputer-Controlled Magnetic-Amplifier Power Supply," pp. 71-75.

M. Shacham, M. B. Cutlip, and P. D. Babcock, "Simulation Package for Small-Scale Systems," pp. 76-83.

Vol. 9, No. 3, Apr. 1985:

B. Heal, "Examination of Microcomputer Interrupt Systems," pp. 107-113.

P. Chaudhuri, "Scheduler for Real-Time Process Control," pp. 114-117.

P. Konnanov and E. Ball, "Real-Time Tracking and Performance Analyzers for Use in Drug Evaluation," pp. 118-123.

J. Doyle, "C—An Alternative to Assembly Programming," pp. 124-132.

N. A. J. Hudson, "Remote Monitoring System Helps Keep Traffic Under Control," pp. 133-137.

Vol. 9, No. 4, May 1985:

B. Heal, "Multiprocessor Solution in OCCAM to an NP-Complete Problem," pp. 162-170.

S. M. Said and K. R. Dimond, "Real-Time High Resolution Data Acquisition Unit for an MC68000 System," pp. 171-178.

B. Srinivasan and H. Gunasingham, "Recoverable File System for Microprocessor Systems," pp. 179-183.

J. S. Saini and E. J. Zaluska, "Performance Analysis of a Distributed Processing System—A Case Study," pp. 184-190.

A. H. Hawkes, "Design Approach to Low-Cost Process Control," pp. 191-199.

Mini-Micro Systems

Vol. 18, No. 6, Apr. 19, 1985:

Spring peripherals digest issue. The four product categories covered are disk drives, printers, tape drives, and graphics terminals.

Vol. 18, No. 7, May 1985:

G. Marini and H. McLarty, "Modula-2 Aids Structured Programming," pp. 77-82.

J. Victor, "Micro-to-Mainframe Choices Expand," pp. 91-108.

C. Morel, "Micro/Mainframe Link Allows Five Sessions," pp. 113-122.

P. Goodrich, "Simple System Approach Increases Throughput," pp. 147-155.

Vol. 18, No. 6, June 1985:

G. R. Talsky, "'Power Pyramid,' IBM Control Market," pp. 85-93.

D. Lytel, "Demand, New Players Boost LAN Market," pp. 97-106.

W. F. Ablondi and L. Lundquist, "IBM, Apple Rule Office Market," pp. 123-132.

Vol. 18, No. 7, June 14, 1985:

Computer digest issue. The four product categories covered are single board, single user, multi-user, and mini.

Mundo Electrónico

No. 149, Apr. 1985:

J. Barbera, "Conferencia Internacional Sobre la 5.a Generación: Últimos Avances Tecnológicos," pp. 43-50.

J. C. Bartolemé, "Sistemas Expertos," pp. 73-80.

F. J. Garijo, "Entornos de Programación," pp. 83-92.

J. Amat, A. Casals, and V. Llario, "Computadores Para el Tratamiento de Imágenes: Estado Actual y Perspectivas Futuras," pp. 95-105.

C. Nadeu and J. B. Mariño, "Comunicación Oral con el Computador," pp. 108-116.

No. 150, May 1985:

J. Agusti and R. López de Mántaras, "Inteligencia Artificial: Técnicas y Actividades Principales," pp. 43-50.

A. S. Cortés, "Robots Inteligentes," pp. 53-63.

M. F. V. Maillo, "Comprensión Automática del Lenguaje Natural," pp. 67-74.

E. M. Pérez and J. B. Piñeiro, "Procesadores Microprogramables (I): Generalidades," pp. 105-112.

J. M. Pérez, P. de la Cruz, and M. C. Costilla, "Bases de Datos Para Minis y Micros," pp. 115-123.

M. P. Vega and P. Zsombor-Murray, "Medición de Presiones de Contacto Distribuidas con μC," pp. 131-133.

F. Serra, M. Bafleur, and J. Buxó, "Tecnología CMOS (II): Procesos Tecnonlógicos y Reglas de Diseño," pp. 135-139.

PC Tech Journal

Vol. 3, No. 5, May 1985:

A. Hansen, "Reflections of Unix," pp. 54-73.

S. Mitchell, "Building Device Drivers," pp. 76-95.

M. A. Covington, "Joystick Metrics," pp. 99-109.

W. H. Murray, "Drawing Circuits," pp. 113-118.

D. Awalt, "Bubble Boards," pp. 123-136.

J. Chumbley, "Testing 1,2,3,4,5," pp. 142-152.

V. Mansfield, "Encryption Methods: Part 2," pp. 157-167.

W. Schreiner, M. Kramer, S. Krischer, and Y. Langsam, "Nonlinear Least-Squares Fitting," pp. 170-190.

T. Forgeron, "Pascal Bugs," pp. 199-204.

PC World

Vol. 3, No. 5, May 1985:

J. J. Hewes, "Gateways to On-Line Services," pp. 148-156.

M. Shinyeda, "DG/One for the Road," pp. 158-162.

J. Getts, "Speaking in Codes," pp. 186-191.

L. B. Stahr, "Tactics for Teleconferencing," pp. 218-225.

"Beginner's Guide." (Previous *PC World* articles on word processing, spreadsheets, database management, communications, graphics, and programming.)

Vol. 3, No. 6, June 1985:

J. Martin, "Dynamic Design," pp. 191-196.

J. Alperson, "Graphics: Before and After," pp. 224-231.

C. Whyte, "Icon Painting," pp. 235-242.

H. Miller, "Show Business," pp. 243-253.

K. Koessel and D. DiNucci, "Clearly Resolved," pp. 256-261.

S. Markle and W. Markle, "Action-Packed Pixels," pp. 274-279.

Vol. 3, No. 7, July 1985:

K. Greenberg, "The DOS Drivers," pp. 122-131.

L. Jordan and J. van der Eijk, "Inside Modems," pp. 144-151.

B. Crider, "Corona at the Speed of Light," pp. 152-160.

D. P. George, "Better and True Basics," pp. 161-167.

J. Goldenberg and K. Koessel, "Advanced Screening," pp. 198-205.

Systems and Software

Vol. 4, No. 5, May 1985:

W. Rauch-Hindin, "23 Internal Unix Tools Are Released by AT&T," pp. 32-34.

M. Chester, "The Boundaries of Unix," pp. 50-62.

"PC Communications Get In On the OSI Act," pp. 65-78.

K. Hughes, "Factory Communications Becoming Standard," pp. 81-86.

J. A. Statx, D. Cerys, and P. Hogan, "Toolkits Speed Software Development," pp. 93-99.

J. Bork and A. Greenspan, "Supermicro Offers Unix System V Virtual Memory," pp. 103-107.

M. Lien and J. P. Malone, "Network Serve Ties Ethernet to SNA," pp. 109-111.

S. B. Russell, "One Card Handles All Data Communications Protocols," pp. 113-115.

Vol. 4, No. 6, June 1985:

R. Bernhard, "Super-Minicomputers—The Hottest Game in Town," pp. 44-58.

E. L. Keller, "Industrial Software Makes CIM Jell," pp. 61-76.

M. Clader, "Dueling Processors Quicken Unix," pp. 79-84.

W. E. Seifert, "Choosing a Transport Protocol," pp. 87-90.

Addresses of Publishers

Business Computer Systems
Cahners Publishing Co.
270 St. Paul St.
Denver, CO 80206
(303) 388-4511

Byte
Byte Publications Inc.
70 Main St.
Peterborough, NH 03458
(603) 924-9281

Computer
IEEE Computer Society
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720
(714) 821-8380

Computer Law & Practice
Frank Cass & Co., Ltd.
Gainsborough House
11 Gainsborough Rd.
London E11 1RS UK

Datamation
Technical Publishing Co.
1301 S. Grove Ave.
Barrington, IL 60010
(312) 774-8115

Dr. Dobb's Journal
People's Computer Co.
2464 Embarcadero Way
Palo Alto, CA 94304
(415) 424-0600

Electronics and *Electronics Week*
McGraw-Hill Publications Co.
1221 Ave. of the Americas
New York, NY 10020
(212) 512-2000

IEEE Communications Magazine
IEEE Electrotechnology Review
IEEE Spectrum
IEEE Transactions (all)
IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854
(201) 981-0060, x133

Macworld
555 De Haro St.
San Francisco, CA 94107
(415) 861-3861

Microprocessors and Microsystems
Business Press International
205 E. 42nd St., Suite 1705
New York, NY 10017
(212) 867-2080

or

Butterworth Scientific Ltd.
Journals Division
PO Box 63
Westbury House
Bury St.
Guildford, Surrey GU2 5BH UK

Mini-Micro Systems
Cahners Publishing Co.
270 St. Paul St.
Denver, CO 80206
(303) 388-4511

Mundo Electrónico
Boixareau Editores, SA
Grand Via de les Corts
Catalanes 594
2 Barcelona 7 Spain

PC Tech Journal
Ziff-Davis Publishing Co.
1 Park Ave.
New York, NY 10016
(212) 725-7947

PC World
555 De Haro St.
San Francisco, CA 94107
(415) 861-3861

Systems and Software
Hayden Publishing Co., Inc.
50 Essex St.
Rochelle Park, NJ 07662
(201) 843-0550

Advertisers & Products

Advertiser

FOR DISPLAY ADVERTISING INFORMATION CONTACT

Southern California and Mountain States: Richard C. Faust Company, 24050 Madison Street, Suite 100, Torrance, CA 90505; (213) 373-9604.
Northern California and Pacific Northwest: Don Farris Company, 161 W. 25th Ave., #102B, San Mateo, CA 94403; (415) 349-2222.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070; (408) 741-0354.
East Coast: Hart Associates, Inc., P.O. Box 339, 42 Lake Blvd., Matawan, NJ 07747; (201) 583-8500.
New England: Arpin Associates, P.O. Box 227, Weston, MA 02193; (617) 899-5613.
George Watts, III, 4 Conifer Dr., Wilbraham, MA 01095; (413) 596-4747.
Midwest: Thomas Knorr, Knorr MicroMedia, Inc. 333 North Michigan Ave. Chicago, IL 60601; (312) 726-2633.
Southeast: Larry C. Shattles, 133 Laurel Oak Drive, Longwood, FL 32779; (305) 788-1950.
Southwest: The House Company, 3817 Richmond Avenue, Suite 110, Houston, TX 77027; (713) 622-2868.

Advertising Director: Mike Koehler, *IEEE MICRO*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-3240, 821-8380.

For production information, conference or classified advertising contact Sandra J. Arteaga, *IEEE MICRO*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, (714) 821-1140.

Products

IEEE MICRO

For further information on advertised products, new products, or literature, fill out the **Reader Service Card** (top). Circle the number on the RS Card that corresponds to the number of the item for which you would like more information.

To indicate your interest in an article or department, fill out the **Reader Interest Card** (bottom). Circle the number on the RI Card that corresponds to the level of interest given in the Reader Interest Survey at the end of the article or department.

Please print or type your name and address.

READER SERVICE CARD

IEEE MICRO INFORMATION ABOUT PRODUCTS

Void after February 15, 1986

8/85

Name _____

Company _____

Address _____

City _____

State _____

Zip _____

Country _____

Title _____

Telephone number ( ) _____

PRODUCTS PURCHASED OR SPECIFIED	FOR JOB	FOR HOBBY
Computers	☐	☐
Peripherals	☐	☐
Data communications equip.	☐	☐
Memories, components	☐	☐
Software and services	☐	☐
Publications	☐	☐
Other	☐	☐

☐ Please send me information on advertising in *IEEE Micro*.

Product announcements, and products for review, should be sent to Richard Landry, Managing Editor, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

Send more information on numbered items:

1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	97
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94	98
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91	95	99
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100

READER INTEREST CARD

IEEE MICRO EDITORIAL RESPONSE

8/85

Name _____

Company _____

Address _____

City _____

State _____

Zip _____

Country _____

Title _____

Telephone number ( ) _____

Comments:

Do you like IEEE Micro's 1986 editorial calendar (see inside front cover)? What specific articles on the topics listed would you like to see?

Continue comments on other side

Reader Interest Survey:

101	116	131	146	161	176	191
102	117	132	147	162	177	192
103	118	133	148	163	178	193
104	119	134	149	164	179	194
105	120	135	150	165	180	195
106	121	136	151	166	181	196
107	122	137	152	167	182	197
108	123	138	153	168	183	198
109	124	139	154	169	184	199
110	125	140	155	170	185	200
111	126	141	156	171	186	201
112	127	142	157	172	187	202
113	128	143	158	173	188	203
114	129	144	159	174	189	204
115	130	145	160	175	190	205

☐ Please send me the IEEE-CS *Publications Catalog*.

☐ Please send me an IEEE Fellow nomination form.

☐ Please send me an IEEE Computer Society membership application.

☐ Please send me an *IEEE Micro* author's guide.

☐ Please send me information about reviewing article submissions for *IEEE Micro*.

This PO box for reader service cards only.

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
Box 24168
Los Angeles, CA 90024
USA

Comments on articles and other editorial matter:

I liked _____

I disliked _____

I would like _____

For Reader Interest Survey, see other side

PLACE STAMP HERE

IEEE MICRO

10662 Los Vaqueros Circle
Los Alamitos, CA 90720-2578
USA

IEEE COMPUTER SOCIETY
Membership Application

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

Join now and receive one of these popular books absolutely free!

Microprocessors and Microcomputers—

All-new third edition — Selected papers from *Computer* and *IEEE Micro* (372 pp.)

Regular nonmember price: $20.00

Selected Reprints in Software—

Selected papers from *Computer* (282 pp.)

Regular nonmember price: $12.00

Offer does not apply to student member applicants.

Check here for your free book. Check one:

☐ *Microprocessors and Microcomputers*
or
☐ *Selected Reprints in Software*

Check box for additional information:

- ☐ Computer Society Membership Brochure
- ☐ Bulk air option (two-to-six-week delivery of publications) for members in Western Europe, Latin America, and the Far East
- ☐ Student Membership Application
- ☐ Publications Catalog

Schedule of Fees (tax deductible)

IMPORTANT: Pay the FULL-YEAR rate if your application is postmarked between September 1, 1984 and February 28, 1985. Pay the HALF-YEAR rate if your application is postmarked between March 1, 1985 and August 31, 1985. Membership expires December 31, 1985.

For applications received:	Half-Year 3/1/85–8/31/85	Full-Year 9/1/84–2/28/85
• **an IEEE member who wishes to join the Computer Society** ..... IEEE membership no. _____	☐ **$5.00**	☐ **$10.00**
• **a non-IEEE member who wishes to join both IEEE and the Computer Society*** ..... Full IEEE members receive *Spectrum*, *The Institute*, and many additional benefits.	☐ **$52.50**	☐ **$90.00**
• **a non-IEEE member who wishes to join the Computer Society only**	☐ **$17.00**	☐ **$34.00**

*Individuals who join both the IEEE and the Computer Society in Regions 7 (Canada) and 8 (Europe, Mid-East, & Africa) may deduct $6 from full-year rates/$3 from half-year rates; in Region 9 (Latin America), deduct $13/$6.50; in Region 10 (Asia & Pacific), deduct $12/$6.

ACM members who join both the IEEE and the Computer Society may deduct $5 from full-year rates/$2.50 from half-year rates.

Publication Options

If, in addition to your automatic subscription to *Computer* Magazine, you would like to receive:

	add:	
• *IEEE Computer Graphics & Applications* (3061)	☐ $11.00	☐ $22.00
• *IEEE Micro* (3071)	☐ $7.50	☐ $15.00
• *IEEE Design & Test of Computers* (3111)	☐ $6.00	☐ $12.00
• *IEEE Software* (3121)	☐ $7.00	☐ $14.00
• *Transactions on Computers* (1161)	☐ $7.50	☐ $15.00
• *Transactions on Software Engineering* (1171)	☐ $8.00	☐ $16.00
• *Transactions on Pattern Analysis & Machine Intelligence* (1351)	☐ $6.00	☐ $12.00
• *Journal on Lightwave Technology* (4301)	☐ $5.00	☐ $10.00
• *Journal of Robotics and Automation* (4401)	☐ $5.00	☐ $10.00
• *Journal of Solid State Circuits* (4101)	☐ $2.50	☐ $5.00
• *Proceedings of the IEEE* (5011) (available only to IEEE members)	☐ $7.50	☐ $15.00

Total amount remitted with this application $_____

☐ Check or money order enclosed. (Make payable to IEEE. Mail to IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.)

☐ VISA ☐ MasterCard ☐ American Express

Charge Card Number _____ Exp. Date Mo. ____ Yr. ____

I hereby make application for IEEE Computer Society membership and, if elected, will be governed by IEEE's and the Society's Constitutions, Bylaws, and Statements of Policies and Procedures.

Full signature _____ Date _____

MAILING ADDRESS

First Name _____ Middle initial(s) _____ Last name _____

Street address _____

City _____ State/Country _____ Zip _____

EDUCATION (highest level completed) Course _____ Degree received _____ Date _____

Name of educational institution _____ Date of birth _____ ☐ Male ☐ Female

OCCUPATION Title or position _____

Firm name _____

Firm address _____

City _____ State/Country _____ Zip _____

Affiliate society membership no. (if applicable) _____ Grade _____

REFERENCE (an IEEE member; if unknown, a managerial person who knows you professionally.)

Name (print in full) _____

Position _____

Street address _____

City _____ State/Country _____ Zip _____

MAIL TO: IEEE COMPUTER SOCIETY, 10662 LOS VAQUEROS CIRCLE, LOS ALAMITOS, CA 90720.

PMCR 885

CALL FOR PAPERS

IEEE MICRO

FEBRUARY 1986

Special issue on semicustom chip technology

Does the future belong to semicustom chips or to mass-produced integrated circuits? The **February 1986** issue of **IEEE Micro** will explore the impact of semicustom chip technology on integrated circuit design. Articles will focus on the technological and economic issues surrounding arrays and cell libraries, as well as the software that supports them.

Send submissions by **October 1** to:
Richard H. Stern
Associate Editor, IEEE Micro
2101 L Street NW, Suite 800
Washington, DC 20037
(202) 785-9700

Please include six copies of your manuscript, or contact the Managing Editor at (714) 821-8380 for information regarding electronic manuscript submission.

Do you need further information on submitting articles to IEEE Micro? Is there a special topic that you want IEEE Micro to cover in depth? Fill out the Reader Interest Card at the back of the magazine.